

FreeMASTER Communication Driver

Serial, CAN and BDM/JTAG communication using FreeMASTER protocol



Contents

Chapter 1 Introduction.....	3
Chapter 2 Description.....	6
Chapter 3 Driver API.....	18
Appendix A References.....	31
Appendix B Revision history.....	32

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 a serial interface between the application and the host PC. The interface covers the native Serial UART communication and CAN communication for the applicable devices.

This driver also supports the packet-driven BDM interface which enables the debugging direct-memory access interface to be used as the FreeMASTER communication device. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

1.1 Driver version 3

This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is V4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit and 64-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. The V3 driver will be ported to the legacy platforms as needed. Reach out to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new serial or CAN communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. The existing code and low-level driver layers may be used as an example when porting to new targets.

NOTE

Using the FreeMASTER tool and FreeMASTER Communication Driver with non-NXP MCU platforms is not permitted by the license terms.

1.2 Target platforms

The new implementation uses the following abstraction mechanisms which simplify driver porting to new MCU platforms and supporting new communication modules:

- **General CPU Platform** (see source code in the */src/platforms* directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs.
- **Transport Communication Layer** - The Serial, CAN, PD-BDM, and other methods of transport logic are now implemented as a driver layer called FMSTR_TRANSPORT with a uniform API. This abstraction enables new communication interfaces to be added easily in future versions (for example; to support Ethernet or TCP/IP communication).
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different

implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The *mcuxsdk* folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The "ampsdk" drivers target automotive-specific MCUs and their respective SDKs. The "dreg" implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

1.3 Replacing existing drivers

For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

1.4 Clocks, pins, and peripheral initialization

The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

1.5 MCUXpresso SDK

The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as an optional "middleware" component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

1.6 Example applications

There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer's physical or virtual COM port. The typical transmission speed is 115200 bps.
- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER 3.1 (or later) to be used with a JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.

- **fmstr_any** is a special example application which fully leverages and demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically. MCUXpresso Config Tools V8 or later are required to support the Serial, CAN, and PD-BDM options. Later versions will support the USB-CDC and other interfaces.

Chapter 2

Description

2.1 Introduction

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

2.2 Features

The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels **(NEW)** .
- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space **(NEW)** .
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— 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.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables **(NEW)** .
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files **(NEW)** .
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling you to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling you to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- TSA support to describe User Resources like external EEPROM or SD Card files.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

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 V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.
- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

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 be shorter than the MTU to 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 driver uses size-aligned operations to read the target memory (for example; proper read-word instruction access when an address is aligned to 4 bytes).

2.2.3 Memory Write

Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; proper read-word instruction access when an address is aligned to 4 bytes).

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 and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

2.2.5 Oscilloscope

The protocol and driver enables any number of variables to be read at once with 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.

You may configure the driver to support any number of Oscilloscope instances to enable simultaneous graphs to be displayed on the host computer screen.

2.2.6 Recorder

The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such 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.

You may configure the driver to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables you to set the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

2.2.7 TSA

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

The TSA feature enables you to create 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.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

2.2.8 TSA Safety

When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses 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.

2.2.9 Application commands

The Application Commands are high-level messages that can be delivered from the PC Host 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 the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. 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 in general.

2.2.10 Pipes

The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. 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 losses.

2.2.11 Serial single-wire operation

The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- "External" single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- "True" single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

2.3 Driver files

The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the `.c` files must be added to your project, compiled, and linked together with your application.
 - *freemaster.h*—master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *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.
 - *freemaster_defcfg.h*—defines the default values for each FreeMASTER configuration option if 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_protocol.c*—implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c*—handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. 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. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as 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. 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. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *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 (via *freemaster.h*).
 - *freemaster_sha.c*—implements the SHA-1 hash code used in the password authentication algorithm.
 - *freemaster_private.h*—contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the driver configuration you provided in the *freemaster_cfg.h* file.
 - *freemaster_serial.c*—implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
 - *freemaster_serial.h*—defines the low-level character-oriented Serial API.
 - *freemaster_can.c*—implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
 - *freemaster_can.h*—defines the low-level message-oriented CAN API.
 - *freemaster_pdbdm.c*—implements the packet-driven BDM communication buffer and other communication-related operations.
- **src/drivers/[sdk]/serial**—contains the code related to the serial communication implemented using one of the supported SDK frameworks.

- *freemaster_serial_XXX.c* and *.h*—implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- *src/drivers/[sdk]/can*—contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h*—implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.

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 before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the 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, 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 a project-specific location, so that it does not affect the other applications that use the same driver.

2.4.1 Configurable items

Table 1 describes the *freemaster_cfg.h* configuration options.

Table 1. Driver configuration options

Statement	Values	Description
<pre>#define FMSTR_LONG_INTR #define FMSTR_SHORT_INTR #define FMSTR_POLL_DRIVEN</pre>	<pre>boolean (0 or 1) boolean (0 or 1) boolean (0 or 1)</pre>	<p>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 Driver interrupt modes.</p> <p>FMSTR_LONG_INTR—long interrupt mode FMSTR_SHORT_INTR—short interrupt mode FMSTR_POLL_DRIVEN—poll-driven mode</p> <p>Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.</p>
<pre>#define FMSTR_DISABLE</pre>	<pre>boolean (0 or 1)</pre>	<p>Define it as non-zero when you want to disable all FreeMASTER features, exclude the driver code, and compile all its API functions empty.</p> <p>The default value is “false”.</p>
<pre>#define FMSTR_TRANSPORT</pre>	<pre>transport interface</pre>	<p>Select the FreeMASTER transport interface. Use one of the pre-defined constants, as implemented by the driver.</p> <p>The current driver supports the following transports:</p>

Table continues on the next page...

Table 1. Driver configuration options (continued)

		<ul style="list-style-type: none"> - FMSTR_SERIAL - serial communication protocol - FMSTR_CAN - using CAN communication - FMSTR_PDBDM - using packet-driven BDM communication
#define FMSTR_SERIAL_DRV	low-level driver interface	<p>When FMSTR_TRANSPORT is defined as FMSTR_SERIAL, then FMSTR_SERIAL_DRV must be configured as the name of a serial low-level driver, implementing the physical communication.</p> <p>When using MCUXpresso SDK, use one of the following constants (see /drivers/mcuxsdk/serial implementation):</p> <ul style="list-style-type: none"> - FMSTR_SERIAL_MCUX_UART - UART driver - FMSTR_SERIAL_MCUX_LPUART - LPUART driver - FMSTR_SERIAL_MCUX_USART - USART driver - FMSTR_SERIAL_MCUX_MINIUSART - miniUSART driver - FMSTR_SERIAL_MCUX_QSCI - DSC QSCI driver - FMSTR_SERIAL_MCUX_USB - USB/CDC class driver (see also the code in the /support/mcuxsdk_usb folder) <p>Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_SERIAL_DRV. For example:</p> <ul style="list-style-type: none"> - FMSTR_SERIAL_DREG_UART - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.
#define FMSTR_SERIAL_BASE	address	<p>Optional. Specify the base address of the UART, LPUART, USART, or another serial peripheral module to be used for the communication.</p> <p>This value is not defined by default. User application should call FMSTR_SetSerialBaseAddress() to select the peripheral module.</p>
#define FMSTR_COMM_BUFFER_SIZE	0 or 32...255	<p>Specify the size of the communication buffer to be allocated by the driver. The default value, which suits all driver features, is used when this option is defined as 0.</p> <p>The default value is "0" (automatic).</p>
#define FMSTR_COMM_QUEUE_SIZE	0...255	<p>Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. Only used with the Serial transport.</p> <p>The default value is 32 B.</p>
#define FMSTR_SERIAL_SINGLEWIRE	0	<p>Set to non-zero to enable the "True" single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.</p>

Table continues on the next page...

Table 1. Driver configuration options (continued)

#define FMSTR_CAN_DRV	low-level driver interface	<p>When FMSTR_TRANSPORT is defined as FMSTR_CAN, then FMSTR_CAN_DRV must be configured as a name of a low-level driver implementing the physical CAN communication.</p> <p>When using MCUXpresso SDK, use one of the following constants (see <i>/drivers/mcuxsdk/can implementation</i>):</p> <ul style="list-style-type: none"> - FMSTR_SERIAL_MCUX_FLEXCAN - FlexCAN driver - FMSTR_SERIAL_MCUX_MCAN - MCAN driver - FMSTR_SERIAL_MCUX_MSCAN - msCAN driver <p>Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.</p>
#define FMSTR_CAN_BASE	address	<p>Optional. Specify the base address of the msCAN, FlexCAN, MCAN, or another CAN peripheral module to be used for communication.</p> <p>This value is not defined by default. Your application should call FMSTR_SetCanBaseAddress() to select the peripheral module.</p>
#define FMSTR_FLEXCAN_TXMB	0...15 (0...31)	<p>Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for the TX communication.</p> <p>The default value is "0".</p>
#define FMSTR_FLEXCAN_RXMB	0...15 (0...31)	<p>Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for the RX communication.</p> <p>The default value is "1".</p>
#define FMSTR_USE_READMEM	boolean (0 or 1)	<p>Define it as non-zero to implement the Memory Read command (recommended).</p> <p>The default value is "true".</p>
#define FMSTR_USE_WRITEMEM	boolean (0 or 1)	<p>Define it as non-zero to implement the Memory Write command (recommended).</p> <p>The default value is "true".</p>
#define FMSTR_DEBUG_TX	boolean (0 or 1)	<p>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.</p> <p>This feature requires the FMSTR_Poll() function to be called periodically.</p>

Table continues on the next page...

Table 1. Driver configuration options (continued)

		The default value is "false".
Oscilloscope		
#define FMSTR_USE_SCOPE	number (>= 0)	Number of Oscilloscope instances to be supported. Set it to 0 to disable the Oscilloscope feature. The default value is 0.
#define FMSTR_MAX_SCOPE_VARS	number (>=2)	Number of variables to be supported by each Oscilloscope instance. The default value is 8.
Recorder		
#define FMSTR_USE_RECORDER	number (>= 0)	Number of Recorder instances to be supported. Set it to 0 to disable the Recorder feature. The default value is 0.
#define FMSTR_REC_BUFF_SIZE	number	Obsolete, kept for backward compatibility only. Defines the size of the memory buffer used by the Recorder instance #0. Default: not defined, user calls the FMSTR_RecorderCreate() API function and specifies this parameter in run time.
#define FMSTR_REC_TIMEBASE	number	Obsolete, kept for backward compatibility only. Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0. Use one of the following macros: FMSTR_REC_BASE_SECONDS(x) FMSTR_REC_BASE_MILLISEC(x) FMSTR_REC_BASE_MICROSEC(x) FMSTR_REC_BASE_NANOSEC(x) Default: not defined, user calls the FMSTR_RecorderCreate() API function and specifies this parameter in run time.
#define FMSTR_REC_FLOAT_TRIG	boolean (0 or 1)	Define it as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library. The default value is "false".
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".

Table continues on the next page...

Table 1. Driver configuration options (continued)

#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".
#define FMSTR_USE_TSA_SAFETY	boolean (0 or 1)	Enable the memory access validation in the FreeMASTER driver. The host cannot access the memory not described by any TSA descriptor. The write access is denied for 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 the 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.

2.4.2 Driver interrupt modes

To implement the serial communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.g* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

See [Table 2](#) for the description of each mode:

Table 2. Driver interrupt modes

Mode	Description
Completely Interrupt-Driven (FMSTR_LONG_INTR = 1)	In this mode, both the communication and the FreeMASTER protocol decoding is done in the FMSTR_SerialIsr, FMSTR_CanIsr, or other interrupt service routine. Because the protocol

Table continues on the next page...

Table 2. Driver interrupt modes (continued)

	<p>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.</p> <p>In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from that handler.</p> <p>Note: The BDM and Packet-Driven BDM interfaces don't support the interrupt mode.</p>
<p>Mixed Interrupt and Polling Modes (FMSTR_SHORT_INTR = 1)</p>	<p>In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the FMSTR_SerialIsr, FMSTR_CanIsr, or other interrupt service routine, while the protocol decoding and execution is handled by the FMSTR_Poll routine. Call FMSTR_Poll during the idle time in the application main loop.</p> <p>The interrupt processing in this mode is relatively fast and deterministic. Upon an serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received packet is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.</p> <p>In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from that handler.</p> <p>When the serial interface is used as the 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_QUEUE_SIZE) and the character time is the time needed to transmit or receive a single byte over the SCI line.</p> <p>Note: The BDM and Packet-Driven BDM interfaces don't support the interrupt mode.</p>
<p>Completely Poll-driven (FMSTR_POLL_DRIVEN = 1)</p>	<p>In this mode, both the communication and the FreeMASTER protocol decoding are done in the FMSTR_Poll routine. No interrupts are needed and the FMSTR_SerialIsr, FMSTR_CanIsr, and similar handlers compile to empty code.</p> <p>When using this mode, ensure that the FMSTR_Poll function is called by the application at least once per the serial "character time" which is the time needed to transmit or receive a single character.</p>

In the latter two modes (FMSTR_SHORT_INTR and 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 with a priority higher than the priority of the communication 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 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. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

2.6 Embedded communication interface initialization

The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is your application code's responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, 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 the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a 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 the short interrupt modes of the driver (see [Driver interrupt modes](#)), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

NOTE

It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

2.7 FreeMASTER Recorder calls

When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. 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 the driver code call the FMSTR_Recorder in the main application loop.

In applications where you call the FMSTR_Recorder periodically with a constant period (equidistantly in a time domain), specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

2.8 Driver usage

Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

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

- Make sure that all *.c files of the FreeMASTER driver from the *src/common andsrc/platforms/#your_platform#* folder are a part of your compiler project. See [Driver files](#) for more details.
- Configure the FreeMASTER driver by creating or editing the *freemaster_cfg.h* file and by saving it into the application project directory. See [Driver configuration](#) for more details.
- Include the *freemaster.h* file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, 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 so that the interrupts can be handled by the CPU.

2.9 Communication troubleshooting

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

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial 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

```
FMSTR_BOOL 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 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.

A 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_protocol.c

Description

In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see [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 task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the FMSTR_Poll function is called at least once per the time calculated as:

$N * Tchar$

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.
- $Tchar$ 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 interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

3.1.3 FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void)
```

```
void FMSTR_CanIsr(void)
```

Declaration

freemaster.h

Implementation

low-level driver C file

Description

This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see [Driver interrupt modes](#)), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

NOTE

In a 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_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg)
```

Declaration

freemaster.h

Implementation

freemaster_rec.c

Description

This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance #0 which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the FMSTR_REC_BUFF_SIZE and FMSTR_REC_TIMEBASE options.

For more information, see [Configurable items](#).

3.2.2 FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex)
```

Declaration

freemaster.h

Implementation

freemaster_rec.c

Description

This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance `recIndex`. If the selected 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 conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down 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).

3.2.3 FMSTR_RecorderTrigger

Prototype

```
FMSTR_RecorderTrigger(FMSTR_INDEX recIndex)
```

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.

3.3 Fast Recorder API

The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

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), it defines the so-called TSA tables in the application. This section describes the macros that must 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 exactly 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.

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 access to the 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](#).
- 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.

NOTE

To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

NOTE

Despite its name, the FMSTR_TSA_STRUCT macro may be also used to describe the union data types.

Table 3. 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	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.

Table continues on the next page...

Table 3. TSA type constants (continued)

FMSTR_TSA_FRAC_Q(m,n)	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 similarly 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 the fmstr://text_files/readme.txt URL
```

```
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 the 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 URLs
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 buffAddr, FMSTR_SIZE buffSize);
```

Declaration

freemaster.h

Implementation

freemaster_tsa.c

Arguments

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

buffSize(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 tsaName, FMSTR_TSATBL_STRPTR tsaType, FMSTR_TSATBL_VOIDPTR
varAddr, FMSTR_SIZE32 varSize, FMSTR_SIZE flags);
```

Declaration

freemaster.h

Implementation

freemaster_tsa.c

Arguments

tsaName(in)—name of the object

tsaType(in)—name of the object type

varAddr(in)—address of the object

varSize(in)—size of the object

flags(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_NON_VAR—other entry, describing structure types, structure members, enumerations, and other types

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 an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See [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* dataLen)
```

Declaration

freemaster.h

Implementation

freemaster_appcmd.c

Arguments

dataLen(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 [FMSTR_GetAppCmd](#)).

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 resultCode)
```

Declaration

freemaster.h

Implementation

freemaster_appcmd.c

Arguments

resultCode(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 `resultCode` 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 resultDataAddr, FMSTR_SIZE resultDataLen);
```

Declaration

freemaster.h

Implementation

freemaster_appcmd.c

Arguments

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

resultDataLen (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 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 `FMSTR_GetAppCmdData` and the data buffer after `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 appCmdCode, FMSTR_PAPPCMDFUNC callbackFunc);
```

Declaration*freemaster.h***Implementation***freemaster_appcmd.c***Arguments****appCmdCode**(in)—the Application Command code for which the callback is to be registered**callbackFunc**(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 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 pipePort, FMSTR_PPIPEFUNC pipeCallback, FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize, FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize, FMSTR_U8 type, const FMSTR_CHAR *name);****Declaration***freemaster.h***Implementation***freemaster_pipes.c***Arguments****pipePort**(in)—port number that identifies the pipe for the client

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

pipeRxBuff (in)—address of the receive memory buffer

pipeRxSize (in)—size of the receive memory buffer

pipeTxBuff (in)—address of the transmit memory buffer

pipeTxSize (in)—size of the transmit memory buffer

type (in)—a combination of FMSTR_PIPE_MODE_xxx and FMSTR_PIPE_SIZE_xxx constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.

name (in)—user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

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 pipeHandle);
```

3.6.2 FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

Declaration

freemaster.h

Implementation

freemaster_pipes.c

Arguments

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

Description

This function de-initializes 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 pipeHandle, FMSTR_ADDR pipeData, FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

Declaration

freemaster.h

Implementation

freemaster_pipes.c

Arguments

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

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

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

writeGranularity(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 *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* 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 pipeHandle, FMSTR_ADDR pipeData, FMSTR_PIPE_SIZE pipeDataLen,
FMSTR_PIPE_SIZE readGranularity);
```

Declaration

freemaster.h

Implementation

freemaster_pipes.c

Arguments

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

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

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

readGranularity(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 readGranularity 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 non-NXP MPU or MCU products.

[Table 4](#) describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Table 4. 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.

[Table 5](#) describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included in the user application indirectly by the *freemaster.h* file.

Table 5. 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.

[Table 6](#) describes the data types used by the FreeMASTER Pipes API:

Table 6. Pipe-related data types

Type name	Description
FMSTR_HPIPE	Pipe handle that 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.

[Table 7](#) describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and they are not available in the application code.

Table 7. 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).

Appendix A

References

- *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](#))
- FreeMASTER tool home: www.nxp.com/freemaster
- FreeMASTER community area: community.nxp.com/community/freemaster
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Appendix B

Revision history

Table 8 summarizes the changes done to this document since the initial release.

Table 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.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 09/2020

Document identifier: FMSTRSCIDRVUG

