FMSTERSCIDRVUG
Rev. 4.0, 04/2019

FreeMASTER Serial Communication Driver

Table of Contents

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 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 referred as 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. To use more advanced FreeMASTER protocol features over the BDM interface, this driver should be used. The driver needs to 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 brings implementation of the new Serial Protocol which significantly extends features and security of its predecessor. The new protocol internal number is V4 and its specification is available in a documentation accompanying the driver code.

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

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

Please note that using the FreeMASTER tool and FreeMASTER Communication Driver with other than NXP microcontroller platforms is NOT permitted by the license terms.

1.2 Target Platforms

Unlike the older versions, the driver is not anymore configurable for a specific microcontroller platform like Kinetis, ColdFire or S12Z. The new implementation uses the following abstraction mechanisms which simplify driver porting:

- **General CPU Platform** (see source code in `/src/platforms` directory). Code in this layer is only specific to native data type size and CPU architecture; for example alignment-aware memory copy routines etc. This driver version brings platform-specific files for Little-endian 32-bit platform. This may be very easily cloned to Big-endian architecture or architecture of different bus sizes. Future versions will bring support for 64-bit platforms and also DSC, S08 and S12Z platforms.

- **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 FlexCAN or MCAN modules.

1.3 Replacing existing drivers

For all supported platforms the driver described in this document replaces the V2 implementation and also older driver implementations which were available separately for individual platforms, and were known as the 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. User application code is responsible for general initialization of clock sources, pin multiplexers and peripheral registers related to communication speed. Such initialization should be done first before calling the `FMSTR_Init` function.

It is recommended to develop the user application using one of Software Development Kits (SDKs) available from 3rd parties or directly from NXP, such as MCUXpresso IDE and related MCUXpresso Tools. This approach will simplify 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 microcontrollers. MCUXpresso Config Tools may be used to generate clock-setup and pin-multiplexer setup code suitable for selected processor.

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

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 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 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 suitable also to access 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 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 data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with. New V4 style of CRC used.
- Layered approach supporting Serial, CAN, PD-BDM and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART and other physical implementations of serial interface, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN and other physical implementations of 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 are available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. An “external” with RX and TX shorted on board as well as the “true” mode when MCU module supports 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 in order to identify the target and be able 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 Maximum size of Transmission Unit (i.e. 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 a password authentication.
- Number of 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 MTU to fit into the outgoing communication buffer. In order 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 (e.g. 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 any RAM memory location on the target device. A single write command frame must be shorter than MTU to fit into the target communication buffer. Larger requests are needed to be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (e.g. 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 is supported by the driver using Read-Modify-Write approach.

Care must be taken when writing bit fields of volatile variables which are also modified in an application interrupt. The interrupt may be serviced in the middle of read-modify-write operation and cause data corruption.

2.2.5 Oscilloscope

The protocol and driver enables any number 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.

User 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.

User 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 enable user to set recording point differently for each instance. For example, one instance may be recording data in a general timer interrupt while other instance may record at a specific control algorithm time in 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 or SD Card files, memory-mapped files, virtual directories, web URL hyperlinks or constant enumerations.

2.2.8 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.

2.2.9 Application Commands

The Application Commands are high-level messages that can be deliver delivered from 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 has been handled, the host receives the Result Code and read 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 implements 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.

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

2.2.11 Serial Single-wire Operation

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

- "External" single-wire operation where Receiver and Transmitter pins are shorted on board. This more is supported by default in the MCU driver as Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for 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 needs to be enabled by defining 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 (e.g. 32-bit Little Endian platform). Each such folder contains platform header file with data types and a code which implements potentially platform-specific operations like aligned memory access.
- ***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.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 default values for each FreeMASTER configuration option in case 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. In case 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. In case 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_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 compile-time verification of the driver configuration provided by user in freemaster_cfg.h file.
  - — *freemaster_serial.c*—implements the serial protocol logic including CRC, FIFO queuing, and other communication-related operations. This code calls the functions of 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 CAn frame and other communication-related operations. This code calls the functions of 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 code related to serial communication implemented using one of supported SDK frameworks.
  - — *freemaster_serial_XXX.c and .h*—Implement low-level access to communication peripheral registers. Different file exists for UART, LPUART, USART and other kinds of Serial communication modules.
- ***src/drivers/[sdk]/can***—contains code related to serial communication implemented using one of supported SDK frameworks.
  - — *freemaster_XXX.c and .h*—Implement low-level access to communication peripheral registers. Different file exists for 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 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 the project-specific location, so that it does not affect the other applications that use the same driver.

2.4.1 Configurable items

The following table describes the *freemaster_cfg.h* configuration 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 2.4.2, "Driver interrupt modes". FMSTR_LONG_INTR—long interrupt mode FMSTR_SHORT_INTR—short interrupt mode FMSTR_POLL_DRIVEN—poll-driven mode **Note:** Some options may not be supported by all communication interfaces. For example FMSTR_SHORT_INTR 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, exclude the driver code and compile all its API functions empty. The default value is “false”.
#define FMSTR_TRANSPORT	transport interface	Select the FreeMASTER transport interface. Use one of the pre-defined constants as implemented by the driver. The current driver supports the following transports: - **FMSTR_SERIAL** ... serial communication protocol - **FMSTR_CAN** ... using CAN communication - **FMSTR_PDBDM** ... using packet-driven BDM comm.
#define FMSTR_SERIAL_DRV	low-level driver interface	When FMSTR_TRANSPORT is defined as FMSTR_SERIAL, then FMSTR_SERIAL_DRV needs to be configured as a name of serial low-level driver implementing the physical communication. When using MCUXpresso SDK, use one of the following constants (see /drivers/mcuxsdk/serial implementation): - **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_USB** - USB/CDC class driver (see also code in /support/mcuxsdk_usb) Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_SERIAL_DRV. For example: - **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	Optional. Specify the base address of the UART, LPUART, USART or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#define FMSTR_COMM_BUFFER_SIZE	0 or 32...255	Specify size of the communication buffer to be allocated by the driver. A default value which suits all driver features will be used when this option is defined as 0. The default value is “0” (automatic).
#define FMSTR_COMM_RQUEUE_SIZE	0...255	Specify size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. Only used with Serial transport. The default value is 32 B.
#define FMSTR_SERIAL_SINGLEWIRE	0	Set to non-zero to enable “True” single-wire mode which uses a single MCU pin to communicate. The low-level driver enables pin direction switching when MCU peripheral supports it.
#define FMSTR_CAN_DRV	low-level driver interface	When FMSTR_TRANSPORT is defined as FMSTR_CAN, then FMSTR_CAN_DRV needs to be configured as a name of low-level driver implementing the physical CAN communication. When using MCUXpresso SDK, use one of the following constants (see /drivers/mcuxsdk/can implementation): - **FMSTR_SERIAL_MCUX_FLEXCAN** - FlexCAN driver - **FMSTR_SERIAL_MCUX_MCAN** - MCAN driver - **FMSTR_SERIAL_MCUX_MSCAN** - msCAN driver Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.
#define FMSTR_CAN_BASE	address	Optional. Specify the base address of the msCAN, FlexCAN, MCAN or other CAN peripheral module to be used for communication. This value is not defined by default. User application should call `FMSTR_SetCanBaseAddress()` to select the peripheral module.
#define FMSTR_FLEXCAN_TXMB	0...15 (0...31)	Only used when FlexCAN low-level driver is used. Define the FlexCAN message buffer for the TX communication. The default value is “0”.
#define FMSTR_FLEXCAN_RXMB	0...15 (0...31)	Only used when FlexCAN low-level driver is used. Define the FlexCAN message buffer for the RX communication. The default value is “1”.
#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_WRITEMEM	boolean (0 or 1)	Define it as non-zero to implement the Memory Write command (recommended). The default value is “true”.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#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	number (>= 0)	Number of Oscilloscope instances to be supported. Set 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 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 size of a memory buffer used by the Recorder instance #0. Default: not defined, user calls FMSTR_RecorderCreate() API function and specifies this parameter in run time.
#define FMSTR_REC_TIMEBASE	number	Obsolete, kept for backward compatibility only. Defines 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 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”.
#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.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#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 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”.

2.4.2 Driver interrupt modes

To implement the serial communication, the FreeMASTER driver handles the Serial or CAN module receive and transmit requests. User uses 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 the following table for a description of each mode:

Table 2-2. 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 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. In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call `FMSTR_SerialIsr` or `FMSTR_CanIsr` functions from that handler. **Note:** The BDM and Packet-Driven BDM interfaces don't support the interrupt mode.
Mixed Interrupt and Polling Modes (FMSTR_SHORT_INTR = 1)	In this mode, the communication processing time is split between the interrupt routine and a 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 the `FMSTR_Poll` during the idle time in the application main loop. 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 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. In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call `FMSTR_SerialIsr` or `FMSTR_CanIsr` functions from that handler. When a serial interface 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. **Note:** The BDM and Packet-Driven BDM interfaces don't support the interrupt mode.
Completely Poll-driven (FMSTR_POLL_DRIVEN = 1)	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. 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.

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 communication interrupt.

2.5 Data types

A 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 user 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 driver implementation layer to upper layers. Other than that, all private variables are all declared as static and are named either with `fmstr_` prefix.

2.6 Embedded communication interface initialization

The FreeMASTER driver does not perform any initialization or configuration of the peripheral module that it uses to communicate. It is user application code responsibility to configure the communication module before the FreeMASTER driver is initialized by the `FMSTR_Init` call.

When a 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 short interrupt modes of the driver (see Section 2.4.2, "Driver interrupt modes"), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to selected serial peripheral module. Make sure to 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 short interrupt modes of the driver (see Section 2.4.2, "Driver interrupt modes"), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to selected CAN peripheral module. Make sure to call the `FMSTR_CanIsr` function from the application handler.

NOTE

It is not necessary to enable or unmask the serial or CAN interrupts 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 > 0), call the `FMSTR_RecorderCreate` function early after `FMSTR_Init` to set up each recorder instance to be used in the application. Then call `FMSTR_Recorder` function periodically in 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 a driver code calls the `FMSTR_Recorder` in the main application loop.

In applications where you call the `FMSTR_Recorder` periodically with a constant period (equidistantly in time domain), specify the period in the recorder configuration structure before calling the `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

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

Follow these steps to enable the basic FreeMASTER connectivity in a user 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 by saving it into application 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.
- 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 `FMSTR_SerialIsr` or `FMSTR_CanIsr` function from this handler.
- Call the `FMSTR_Init` function early in the application initialization code.
- Call the `FMSTR_RecorderCreate` functions for each Recorder instance to enable the recorder feature.
- 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 so interrupts can be handled by the CPU.

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 a 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 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 poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see Section 2.4.2, "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 an "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:

$$N * T_{char}$$

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 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 interrupt processing code of the FreeMASTER driver. In long or short interrupt modes (see Section 2.4.2, "Driver interrupt modes"), this function must be called from an application interrupt service routine registered for communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, and application should register a handler for all vectors and call this function for 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 one recorder instance and enables it to be used by a PC Host tool. User needs to call this function for all recorder instances from 0 to the maximum number defined by 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 freemaster_cfg.h configuration file defines FMSTR_REC_BUFF_SIZE and FMSTR_REC_TIMEBASE options.

See more information in Section 2.4.1, "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 one sample of the variables being recorded using the FreeMASTER Recorder instance `recIndex`. If a 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 any of trigger conditions 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

Fast recorder feature is not available in FreeMASTER driver version 3. This feature was heavily dependent on target platform and was only available for DSC 56F8xxxx.

3.4 TSA API

When 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 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 the 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 3-1.
- `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.

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 3-1. 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 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 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).

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)

Appendix B Revision History

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

Table 3-6. 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.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: FMSTRSCIDRVUG
Rev. 4.0
04/2019