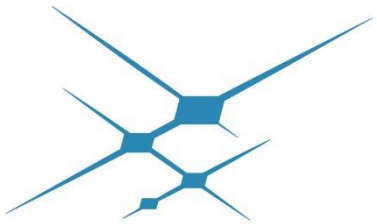
AIS Command Messaging Specification

Rev 1.01



SKYWORKS®

SKYWORKS SOLUTIONS INCORPORATED
1600 NE COMPTON DRIVE, SUITE 300 | HILLSBORO, OREGON 97006, U.S.A.
MAIN +1.503.718.4100 | WWW.SKYWORKSINC.COM

Table of Contents

Table of Figures

Table of Tables

1 Overview

This document discusses the SDK command messaging protocol used to communicate with AIS processors. This messaging protocol was initially utilized with the AV34xx and AV35xx series devices, and now has been utilized for the AV63xx and SKY7635x devices.

This messaging protocol is designed to implement a low overhead query/response protocol to interact with the part from the host system. Another target of the design is to enable communications through any of the communication ports exposed to an external host utilizing variable length packet sizes for communications.

1.1 Conceptual Model/Goals

The model used for AIS Command Messaging communications is a query/response interaction. And is based on a messaging protocol and format, which utilizes the standard set of opcodes described in this document. Some opcodes define messages that include an optional data buffer to be associated with it.

The AIS Command Messaging protocol is defined on a query/response mechanism, where each communication port only processes a single command at a time. This helps to limit the requirements of the embedded implementation and avoids any complexity associated with tracking and managing multiple data buffers. Additionally, the requirement for each message to have a response (including a result status) inherently creates a flow control system and prevents any possibility for buffer overflows or droughts.

The AIS command message header is based on a 12-byte data structure (refer to section 2). This header structure is utilized to either entirely encompass the command and response interactions, or as a header to parameterize the usage of the (optional) associated data buffer. The header includes a *BufferFlag* field, to explicitly indicate if a data buffer is associated with the command or not.

The command message structure is implemented using a standardized format, and a standard set of opcode values. Each message header structure also incorporates information to address different software modules internal to the system.

A Port communicates with an external host or device. A driver Module is associated with each Port.

A Module is an internal service or software interface provided by a device.

2 Message Header Structure

The structure of the 12 byte message header is defined in Table 1.

Table 1: Message Header Structure

Byte Offset	Start Bit	Field	Size (bits)	Description
0	0	Source	8	The source module identifier.
1	0	Dest	8	The destination module identifier.
2	0	Opcode	6	The opcode for the operation.
2	6	Reserved	1	Reserved. Must be 0.
2	7	BufferFlag	1	The flag to indicate if a buffer is being used.
3	0	Status	6	The status of the operation (on return).
3	6	Type	2	The type of the message.
4	0	Param0	32	The first parameter for the command.
8	0	Param1	32	The second parameter for the command.

Each of these fields is detailed as follows:

Table 2: Message Header Field Definitions

Field	Details
Source	Identifies the source module of the command. This field identifies the module that the message was received from. The set of standard modules supported by the AIS Command Messaging protocol are defined in Table 5.
Dest	Identifies the destination module of the message. This field enables the message router to direct the message to the appropriate module. The set of standard modules supported by the AIS Command Messaging protocol are defined in Table 5.
Opcode	Indicates the operation that the command is requesting. The set of standard opcodes are defined in Table 3. *NOTE: Modules are independent of each other and can reuse the various opcode values for their own operation*.
BufferFlag	Flag to indicate if a data buffer is associated with the message. **Value** — **Description** 1 — A data buffer is associated with the message. The value of Param0 shall define the Address/Offset of the data, and the value of Param1 shall define the length of the data buffer. 0 — No data buffer is associated with the message. The value of Param0 shall define a Parameter Identifier or Request specific data, and the value of Param1 shall define a Parameter Value or Request specific. The size of the data value is operation dependent and may be up to a 32-bit. The BufferFlag must be accurately set as a function of the selected command, otherwise various actions in the protocol processing may result in the value of Param1 being modified.

Field	Details
Status	Indicates the status of the message. This value is only meaningful during the response phase of the communications and should be set to zero on the query phase. The response status values are based on the *ResultStatus* enum: **Value** **Name** **Description** 0 ResultStatus_Success Indicates a successful message execution. 1 ResultStatus_Failed Indicates a general operation failure. 2 ResultStatus_Pending Indicates that the operation is still pending. 3 ResultStatus_NotFound The requested destination module was not found. 4 ResultStatus_NotSupported The requested operation is not supported by the destination module. 5 ResultStatus_ExceededLimits The requested operation exceeded the limits. 6 ResultStatus_NotReady The destination module is not ready. 7 ResultStatus_CommFailure A comms driver has detected a failure. 8 ResultStatus_InvalidFile The file format doesn't match the appropriate settings. 9 ResultStatus_NoCal The chip calibration has not been applied. 10 ResultStatus_OS_Error An OS (MQX) API call failed. 11 ResultStatus_Unauthorized The requested operation has not been authorized. 12 ResultStatus_Busy The requested operation failed because the destination module is busy. Convenience namings. 0 ResultStatus_Complete = ResultStatus_Success. 2 ResultStatus_Continue = ResultStatus_Pending.
Type	Indicates the type or phase of the communications for the command. **Bit Field** **Description** 0b00 Indicates that the message is a result. 0b01 Indicates that the message is a query (inbound). The other values of this field are reserved and shall not be used.
Param0	The first parameter value for the message, or Data Buffer Address/Offset.
Param1	The second parameter value for the message, or Data Buffer Length.

All commands are implemented using this structure.

2.1 Standard Module Values

Refer to the Appendix, Table 5.

2.2 Standard Opcode Values

Table 3 lists the standard Command opcode values currently supported. Any values not listed in this table are reserved and should not be used.

Table 3: Standard Opcode Values

Opcode	BufferFlag	Value	Param0	Param1
CmdOpcode_Read	1	1	Address/Offset	Length (in bytes)
CmdOpcode_Write	1	2	Address/Offset	Length (in bytes)
CmdOpcode_SetParameter	0	3	Parameter Identifier	Parameter value
CmdOpcode_GetParameter	0	4	Parameter Identifier	Parameter value (return)
CmdOpcode_Request	0	5	Various	Various

The following sections will use the Memory, Flash, and File Modules as examples for usage of the OpCodes. Other modules may support these OpCodes. See their respective documentation for details.

2.2.1 Read/Write Details

The CmdOpcode_Read and CmdOpcode_Write operations use a similar format. Each of these commands utilizes a data buffer and adhere to the following rules:

- The Param0 value shall indicate the address or offset for the operation. The meaning of this value will be dependent on the module being addressed.
- The Param1 value will indicate the size (in bytes) of the data buffer to use.

  The size of the internal buffer allocated by a port driver varies between modules. The Param1 value may be modified to truncate the buffer size for the specific port being referenced.

  Additionally, the handler may shorten the length value to indicate an operation has been stopped before the desired length. See example in the section 3.1.

2.2.1.1 Read Details

The Memory, Flash, and File Modules support the CmdOpcode_Read opcode.

2.2.1.1.1 Memory Peek

A Memory Module peek is defined using a CmdOpcode_Read opcode, setting Param0 to the start of the memory address to be read, and Param1 to the number of bytes to be read.

Upon the successful completion of a CmdOpcode_Read, the buffer contains the results of Peek operation.

2.2.1.1.2 Flash Read

A FLASH Module read is defined using a CmdOpcode_Read opcode, setting Param0 to the start of the Flash address to be read, and Param1 to the number of bytes to be read.

Upon the successful completion of a CmdOpcode_Read, the buffer contains the results of FLASH read.

2.2.1.1.3 File Read

A File Module read is defined using a CmdOpcode_Read opcode, setting Param0 to the start of the file offset to be read, and Param1 to the number of bytes to be read.

Upon the successful completion of a CmdOpcode_Read, the buffer contains the results of the File read operation.

NOTE: A File Request must be used to locate the file to be read prior to reading file contents (refer to section 2.2.4.3 for details on locating a file).

2.2.1.2 Write Details

The Memory and Flash Modules support the CmdOpcode_Read opcode.

2.2.1.2.1 Memory Poke

A Memory Module Poke is defined using a CmdOpcode_Write opcode, setting Param0 to the start of the memory address to be written, and Param1 to the number of bytes to be written.

Upon the successful completion of a CmdOpcode_Write, the buffer containing the data to be written will have been transferred to the memory address block pointed to by Param0, up to the number of bytes defined by Param1.

2.2.1.2.2 Flash Write

A Flash Module write is defined using a CmdOpcode_Write opcode, setting Param0 to the start of the Flash address to be written, and Param1 to the number of bytes to be written.

Upon the successful completion of a CmdOpcode_Write, the buffer containing the data to be written will have been transferred to the Flash address block pointed to by Param0, up to the number of bytes defined by Param1.

NOTE: The Flash must be erased to prior to the writing. Refer to section 2.2.2.2.

2.2.2 SetParameter Details

The set parameter opcode is used to control parameter values of the specific module that is being targeted. For the set operation, the Param1 value provides the value being set.

The Memory and Flash Modules support the CmdOpcode_Read opcode.

2.2.2.1 Setting the CPU Speed

The CPU speed is set by issuing a command to the Memory Module using a CmdOpcode_SetParameter opcode, and setting Param0 to the Parameter ID CmdParameter_SetCPUSpeed, and Param1 to the CPU speed value to be set. Refer to Table 6 for a list of the CPU speed values.

2.2.2.2 Setting the FLASH erase address

The Flash Module erase address is set by issuing a command to the Flash Module using a CmdOpcode_SetParameter opcode, and setting Param0 to FlashP_EraseAddress, and Param1 to the Flash address value to be erased.

NOTE: FlashP_EraseAddress MUST be set before setting FlashP_EraseSize.

The Flash erase size is set by setting Param0 to FlashP_EraseSize, and Param1 to the size value to be erased (e.g., size <= 0x1000 erases 4k, size <= 0x10000 erases 64k, size 0 or greater than 0x10000 erases the entire Flash). Unless you're erasing the entire Flash, it's only possibly to erase a single 4k or 64k chunk of Flash at a time, so multiple Flash erase commands may be necessary.

2.2.3 GetParameter Details

The get parameter opcode is used to read parameter values of the specific module that is being targeted. For the get operation, the Param1 value is not used on input and returns the value for the parameter (in the case that the message status indicates success).

The Memory and Flash Modules support the CmdOpcode_Read opcode.

2.2.3.1 Memory Module Parameters

The following commands are issued to the Memory Module using a CmdOpcode_GetParameter opcode.

2.2.3.1.1 CPU speed

The CPU speed is read by setting Param0 to CmdParameter_SetCPUSpeed.

Upon a successful completion, Param1 will contain the CPU speed value. Refer to Table 6 for the CPU speed values.

2.2.3.1.2 Boot Flag

The Flash Boot flag is read by setting Param0 to CmdParameter_FlashBootFlag.

Upon the successful completion of the command, Param1 will contain a value of 1 if an application was loaded from Flash, a value of 0 if not.

2.2.3.1.3 Cache Status

The power on reset status cache is read by setting Param0 to CmdParameter_PorStatusCache.

Upon the successful completion of the command, Param1 will contain the Cache Status value. If bit 0 of Param1 is set, then it indicates “recovery startup” (no flash). Bit 1 set indicates debug mode and bit 3 set indicates a watchdog timer fault.

2.2.3.1.4 Assert Value

The assert value is read by setting Param0 to CmdParameter_AssertValue.

Upon the successful completion of the command, Param1 will contain the address of the last system assert.

2.2.3.1.5 CFS Offset

The CFS offset is read by setting Param0 to CmdParameter_CfsOffset.

Upon the successful completion of the command, Param1 will contain a value indicating the index of the 4k sector in flash that is currently being executed from.

2.2.3.1.6 Image Index

The image index is read by setting Param0 to CmdParameter_ImageIndex.

Upon the successful completion of the command, Param1 will contain a value indicating the index of the image that is currently being executed.

2.2.3.1.7 Recovery Mode

The recovery mode is read by setting Param0 to CmdParameter_RecoveryMode.

Upon the successful completion of the command, Param1 will contain a value indicating the current recovery mode. Recovery mode values are: normal (0), software (1), rom halt (2), and SPIS boot (3).

2.2.3.2 Flash Module

The following command is issued to the Flash Module using a CmdOpcode_GetParameter opcode.

2.2.3.2.1 Flash Status

The Flash status is read by setting Param0 to Status.

Upon the successful completion of the command, Param1 will contain the Flash status value. If bit 0 of Param1 is set, it indicates the flash is busy; if bit 1 is set, it indicates the flash is write enabled.

2.2.4 Request Details

The CmdOpcode_Request operation is unique for each module and does not have a standard meaning assigned to it. The Param0 and Param1 values are flexible and their meaning is defined by the targeted module.

The Memory, Flash, and File Modules support the CmdOpcode_Read opcode.

2.2.4.1 Memory Module

Handles the CmdOpcode_Request operation by treating Param0 as the address of a function to call and Param1 as the parameter that function takes. The called function has no return value.

NOTE: Great care must be taken when using this operation to avoid passing invalid function information to the system.

2.2.4.2 Flash Module

Handles the CmdOpcode_Request operation by returning the Flash status in Param1. If bit 0 of Param1 is set, it indicates the flash is busy; if bit 1 is set, it indicates the flash is write enabled.

2.2.4.3 File Module

Handles the CmdOpcode_Request operation by using Param0 as the index (offset) into the file system to look for a table of contents entry.

If the command was successful, then the requested file was found and Param0 will contain the table of contents size, and Param1 will contain the table of contents flags in the upper word and table of contents type in the lower word.

An unsuccessful command indicates that the requested file is outside the range of files in the system.

It's possible to locate a file by using the CmdOpcode_Request operation and starting the index (offset) at 0. On success, the table of contents flags and type may be interrogated to determine if it is the file you were looking for. If not, increment the index (offset) and send the CmdOpcode_Request again. Continue repeating this operation until the file you are looking for is located, or the end of the file system is reached (indicated by a non-successful CmdOpcode_Request).

2.3 Module/Handler Independence

The opcodes that are defined are interpreted relative to the module handler they are addressed to. This means each of the operations are independent and end up creating independent address and parameter identifier values for each module in the system.

3 Host Interface

The host interface interacts with the embedded platform with messages using the format described in this specification. This interaction operates using a query/response interaction. For each command message that is sent to the system, there is a matching response read back to determine the result status of the command message. For each command, the data buffer contents are provided, or read back as appropriate. The details of the data buffer transfers are provided in section 5.

3.1 PC platform specifics

On PC systems, the software component that implements the low-level communications support is through the AvServe program. AvServe supports various communication port interfaces and exposes an ASCII interface through a network socket server interface. The AvServe program manages these interactions to ensure the query/response cycle is completed for one client without any interactions from commands sent by any other client. This allows multiple client programs to operate concurrently while still having reliable communications to the device.

For commands that use data buffers, the host processing of the message is somewhat more complex. The host interface attempts to issue as few commands as possible to implement a client request. However, when faced with large read or write operations, the command issued from the client many be split between multiple buffers. This behavior adapts to the buffer size limits of the interface communication port.

As an example, a client program attempts to read N bytes of memory from the device at address A. AvServe will attempt to handle this using a single read command. However, the embedded processing in the device will cause this to be limited by the buffer size (X bytes in this example) of the communication port and the return value will indicate this to AvServe. If X < N, the read operation is incomplete at this stage and AvServe will issue a follow on read request of N-X bytes at address A+X. Again, the device could return a possibly short response N bytes long and cause AvServe to read N-2X bytes at address A+2X. This process continues until the read request is no longer than X bytes and the initial request is completed.

In this manner, the host interface is abstracted from the packet size limitations of the communication port.

4 Embedded Implementation

The embedded components are implemented in 3 parts: communication port handlers, a central message dispatcher, and module port handlers. The communication port handlers supported are configured when the associated port drivers are started on the device. The message dispatcher is initialized and started by calling the cmdInit function. This function takes a pointer to a table that lists Module Identifiers and their associated handler functions. The table provided to the cmdInit function allows support for a variable number of standard and application module handlers.

The Module Identifier handler functions are called as commands and are processed out of the command queue. The appropriate Module Identifier handler function is called based on the command destination.

Rules

These rules must be followed for all interactions regardless of the target module or communication port used.

1. Command handlers operate synchronously. This means that they control the command handler thread (which may be the only task level thread) until they complete and will hold up any other commands during that time. This means that they should complete in a reasonable amount of time. If you have a command that takes a long time to complete (for example, erase an entire flash), then you need to break it up such that one command will start the operation and a second command is used to check on the completion status of the operation.
2. All commands MUST accurately indicate if a data buffer is associated with the command. This is necessary for the embedded processing to function correctly.
3. When read/write opcodes are handled by a module, the address space is defined in the context of each module. In this address space the operations must be setup so that the requests can be split at arbitrary address values. This is necessary to support the addressing subdivisions that may result from the limitations imposed by port buffer sizes. This further means that the embedded code must be able to reassemble extended requests as they are received if necessary. See the example in PC platform specifics, earlier in this document.

5 Port Specific Details

Each of the communication ports utilizes different transfer mechanisms to support command transfers. The details for these interactions are defined in this section. If the AvServe host program is used, all the details related to these transfers are handled internally. These details are only important if a custom host interface is being developed.

5.1 USB Port

USB communicates command messages through transfers of the HID Feature reports. Two reports are defined to handle this port.

5.1.1 Command Messages

Feature report ID 0xF0 is used to transfer the base message structure of the commands, and feature report ID 0xF1 is used to transfer the data buffer values associated with these commands. For the base message command structures, the report length is 14 bytes. The first byte of the report is the report ID value (0xF0). For the SetFeature operation the second byte of the report is set to 0. The following 12 bytes contain the data for the command message. To read back the device response, the host will execute a GetFeature request of the 0xF0 report ID. The packet returned will indicate that data is present by setting the second byte of the report to 1. If this byte is returned as 0, the device has not responded to the request yet and the host must poll again. For the 0xF1 report ID, there are a total of 257 bytes consisting of the 0xF1 report ID followed by 256 data bytes.

5.1.2 Command Exchanges

There are three command exchanges that can be utilized:

- Commands without a data buffer – For this case the host will issue a SetFeature (0xF0) to send the command and then request a GetFeature (0xF0) until it receives the associated response.
- Commands writing a data buffer – In the case of a write command, the host will first issue a SetFeature (0xF1) to set the data buffer associated with the command. Once this is completed, the host will perform the SetFeature (0xF0) of the base command to execute the write operation on the device and then request a GetFeature (0xF0) until it receives the associated response.
- Commands reading a data buffer – In the case of a read command, the host will first issue a SetFeature (0xF0) to send the read command and then request a GetFeature (0xF0) until it receives the associated response. Once this is successful, the data buffer for the operation is read by issuing a GetFeature (0xF1) request.

NOTE: All Command Exchanges are only completed when a GetFeature (0xF0) is read and the second byte is set to 1. This may require multiple requests of this report to complete, but it is recommended to timeout after 3 seconds of retries.

5.2 SPI Slave Port

The SPI Slave module uses a standard 4-wire connection. The pin definitions are MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK (Serial Clock), and SSEL (Slave Select). Since the protocol implements both inputs (MOSI) and outputs (MISO), bi-directional data is transferred simultaneously with each clock transition.

5.2.1 SPI Slave General Operation

The SPI Slave transfer is built around a hardware SPI shifter which simultaneously reads in data on the MOSI pin and writes out data on the MISO pin. The management of SPI transfers is done by a host SPI master. Host SPI Master writes may interrupt the slave, but the slave has no interrupt to the host. Information from the slave to the host is discovered through polling. Any higher-level protocol giving meaning to the data that is transferred over the SPI is managed on an upper layer that is not affected by the SPI hardware.

Host SPI Master write operations consist of a command of 0x00 followed by a SPI MOSI transaction. A command of 0x00 can be followed by a SPI MOSI transaction when the SPI status reports that the SPI Rx Data Full status bit RxDataFull is clear. This indicates that the slave has read any data into the SPI Rx Data register and has cleared the SPI Rx Data Full status bit RxDataFull of the SPI Status register. Refer to sections 5.2.2 and 5.2.3 for more details.

SSEL must be de-asserted between transactions for at least the time of one full SCLK period. SCLK need not toggle while SSEL is de-asserted between transactions. SSEL being de-asserted will end a SPI MOSI transaction.

If the SPI Rx Data register is already full, then the new data will be discarded, and the SPI Error bit SPIERR of the SPI Status register is set. The SPI Error bit SPIERR is cleared on the next successful SPI operation.

A host read operation consists of a command of 0x00 followed by a SPI MISO transaction. A command of 0x00 can be followed by a SPI MISO transaction when SPI status reports that the SPI Tx Data Ready Status bit is set. If the host does not wait for SPI Tx Data Ready Status to go active high in the SPI Status byte, then it will not know if the data it reads back is valid. The SPI Tx Data Transaction can be a variable number of bits in length from 8 to the SPI command buffer length (depends on chip and can be variable). SSEL must be de-asserted between transactions for at least the time of one full SCLK period. The rising edge of SSEL will end the transaction and cause SPI Tx Data Ready Status to clear.

Data sent into the SPI Slave causes an interrupt when the internal buffer is full or the SSEL line is de-asserted and is handled by the SPI driver immediately. For data coming out of the SPI Slave hardware, the host must poll for data available.

5.2.2 SPI Slave Mode

There are two SPI configuration parameters, CPOL (Clock Polarity) and CPHA (Clock Phase), which specify which edges of the clock signal are used to sample and drive data. Our slave devices require the master to use CPHA = 1 (SPI transfer mode) and CPOL = 0 (falling edge data latch, rising edge data shift).

5.2.3 SPI Slave Status

The SPI Slave hardware outputs a status byte (see Table 1: Status Byte Encoding). During each transmit operation. If one wants to query status, a command of 0x00 is sent. The status response byte is bit encoded. The host should check the status prior to sending any data into the slave.

Table 4: SPI Slave Status Encoding

BIT No.	Bit Description	Default State	Description
0	RxDataFull	0	0 = Ready to Receive Data 1 = Data buffer is full
1	TxDataReady	0	0 = Buffer empty 1 = Data available to Transmit

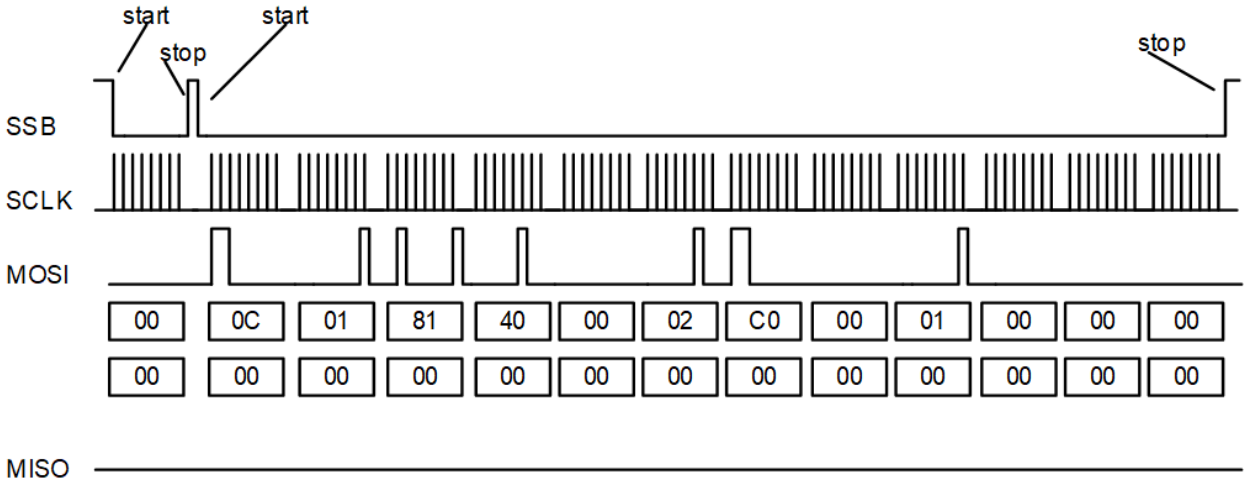
BIT No.	Bit Description	Default State	Description
2	SPIERR	0	0 = No Error 1 = Overflow of RX data
3	Reserved	0	Unused
4	SW Busy	0	0 = Ready to receive SPI message. 1 = Internal Firmware is busy.
5	Reserved	0	Unused
6	SW Error	0	0 = Normal operation. 1 = SW Assert occurred.
7	Reserved	0	Unused

5.2.4 SPI Slave Receive (Master to Slave)

To send data to the slave, the first byte of the transaction must be ModuleId_SPIS (the source).

If the SPI Rx Data Full status bit RxDataFull is clear, one byte can be sent into the device. It is necessary to check the response Status byte after each transaction before continuing with the next byte. Continue sending bytes until the transaction is complete. It is necessary to deactivate SSEL after each transaction.

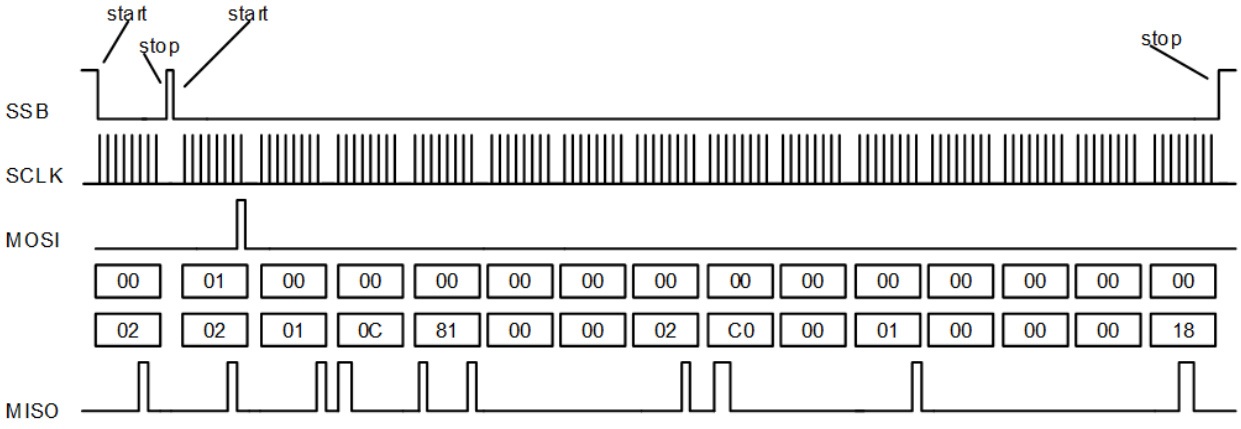
Figure 1: SPI Slave Receive Transaction



5.2.5 SPI Slave Transmit (Slave to Master)

If data is waiting to be sent, bit one of the Status Byte (TxDataReady) will be set. Sending a command of 0x01 will initiate a transfer from the SPI slave to the host. As the 0x01 is clocked in a status byte will be clocked out. Subsequent bytes (either 0x00 or 0x01) clocked in will result in data bytes being clocked out. Those subsequent bytes sent to clock out the data bytes are taken by the SPI slave hardware; not passed onto the SPI slave firmware driver. To terminate a transfer, deactivate the SSEL signal. If the clocking out of data exceeds the available data in the buffer, the data returned will be either the last byte read, repeated, or it will be 0x00 and can be ignored.

Figure 2: SPI Slave Transmit Transaction



6 APPENDIX A

6.1 Module Values for SKY7635x Devices

The following table lists the module values supported for SKY7635x devices.

Any values not listed here are reserved and should not be used.

Table 5: Module Values

Module	Value [dec, 0x]	Description
Memory	1, 0x01	The memory module.
Register	2, 0x02	The register module.
SPIM	4, 0x04	The SPI Master module.
File	5, 0x05	The FLASH file system module.
Flash	6, 0x06	The FLASH memory module. Direct FLASH access.
TWIM	7, 0x07	The TWI (I2C) master Port.
USB	8, 0x08	The USB Port.
JTAG	9, 0x09	The JTAG Port.
Log	10, 0x0A	The LX logging module.
UART	11, 0x0B	The UART Port.
SPIS	12, 0x0C	The SPI slave port.
TWIS	13, 0x0D	The TWI (I2C) slave port.
Audio	14, 0x0E	The audio module. (audport)
DVM	17, 0x11	The DVM module.
NVM	18, 0x12	The NVM module.
Upgrade	20, 0x14	The Firmware Upgrade module.
SysPower	22, 0x16	The system power and clocking module.
MAC	26, 0x1A	The MAC module.
MsgRoute	27, 0x1B	The message router module.
SysCfg	28, 0x1C	The system configuration module.
Application	64, 0x40	The Application module.
Host	66, 0x42	A command that gets handled by the remote "host", (i.e. the smsocket program).

6.2 CPU Speed Values for SKY7635x Devices

The following table lists the CPU speed values defined for the system. Any values not listed here are reserved and should not be used.

Table 6: CPU Speed

CPU Speed	Value
1.5 MHz	0
3 MHz	1
6 MHz	2
12 MHz	3
24 MHz	4
48 MHz	5

7 Revision Information

ACT Revision	Date	Change
1.00	7/9/20	- Initial Release
1.01	12/14/20	- Terminology and format corrections.

8 Contact Information and Legal Disclaimer

Skyworks Solutions Incorporated, Artificial Intelligence Solutions

1600 NE Compton Drive, Suite 300

Hillsboro, Oregon 97006

U.S.A.

Main: +1.503.718.4100

Fax: +1.503.718.4101

www.skyworksinc.com

Customers are responsible for their products and applications using Skyworks products, which may deviate from published specifications as a result of design defects, errors, or operation of products outside of published parameters or design specifications. Customers should include design and operating safeguards to minimize these and other risks. Skyworks assumes no liability for applications assistance, customer product design, or damage to any equipment resulting from the use of Skyworks products outside of stated published specifications or parameters.