AN4353
Application note

SPC57xx/SPC58xx: Debug over CAN

Introduction

The SPC57xx/SPC58xx family of multicore 32-bit microcontrollers is initially intended for automotive power train applications. It is based on e200 Power Architecture® cores.

This application note describes SPC57xx/SPC58xx the new ‘Debug over CAN’ feature that is implemented on those devices and its implementation is given, background information about the involved modules (MCAN, DMA and JTAGM) is provided, and the sequence of operation is described on a high level. This application note explains how debug messages are handled by the MCAN module. The DMA data transfers between the MCAN and the JTAGM modules are also detailed along with how JTAG traffic is generated. A practical example to implement the theory from the application note is provided. A glossary of terms and definitions is provided in the appendix.

‘Debug over CAN’ is a new additional feature of the SPC57xx/SPC58xx microcontrollers.The feature is intended to allow debugging of ECUs in the field, where standard debug interfaces like JTAG or Nexus are usually not readily available but access to the CAN interface is possible.

This application note is intended to:

- Explain the feature to the user and give a thorough understanding to enable engineers to develop applications around it
- Detail the capabilities and operation of Debug over CAN
- Supply example application level initialization code to enable the feature
- Supply example CAN debug traffic to allow tool developers to test the feature

Contents

List of tables

List of figures

1 Overview

'Debug over CAN' is a new feature that will be implemented on various devices of the SPC57xx/SPC58xx family of microcontrollers targeted towards automotive applications. *Table 1* gives an overview of the implementation on different devices:

Table 1. Device list

Devices	
SPC57EM80 (cut2)	SPC582B60 CH1M
SPC574K72 (cut2)	SPC58EC80 CH4M
SPC572L64 (cut1)	

'Debug over CAN' allows debug of application hardware where access to the dedicated JTAG/LFAST interface signals is not easily available. The debug over CAN mechanism is intended to provide basic debug functionality with some reduction in bandwidth and features compared to use of the dedicated JTAG or LFAST interfaces.

Use of the CAN interface for debug purposes requires the use of the following resources to be exclusively used for that application:

- MCAN resources: 3 CAN filters need to be configured for the 3 CAN debug messages. Furthermore two dedicated TX elements are needed for the transmission of the debug messages that are to be sent back to the external tool
- DMA resources: a total of six DMA transfers is required per Debug over CAN cycle. When using the scatter/gather mechanism of the DMA module only one DMA channel is needed, plus 192 byte of flash memory (32 byte for each of the 6 required Transfer Control Descriptors)

Initialization of both MCAN and DMA via software is required, but once this initialization has completed, debugging over CAN is possible without any further software overhead. The initialization can be performed in the boot code.

As the debug over CAN scheme generates internal JTAG messages based on received CAN data, all JTAG clients and included debug resources are accessible. Basic device trace capability is also possible by configuring the device trace hardware to stream to a device overlay/trace RAM, which can be read later using debug over CAN.

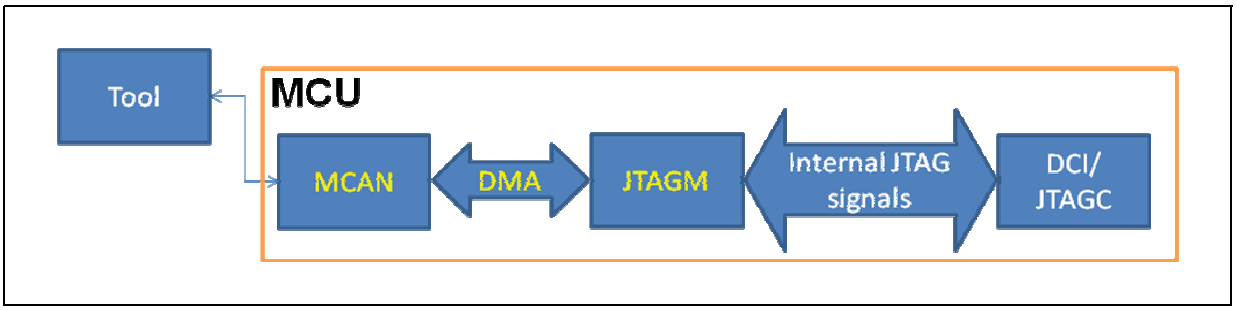
Note: No other (external) JTAG tool / debugger can be connected when using 'Debug over CAN'. If a JTAG tool is connected, the JTAGM is not able to access the DCI resources.

The debug over CAN scheme supports the following features:

- Operation through MCAN_1 or MCAN_2 interfaces
  - Makes use of MCAN debug enhancements
  - Debug message selected by filter configuration SFEC/EFEC=111
- Allows debug of hardware where JTAG access is not available
- Debug traffic carried over application CAN bus
- Debug traffic on CAN coexists with application traffic
- No software overhead after initialization
- Flexible selection of CAN identifiers for debug use
- Access to all JTAG debug facilities, including CPU run control

Figure 1 gives a high-level overview of the 'Debug over CAN' sequence and the modules that are involved.

Figure 1. Debug over CAN sequence



In a typical application in the field, an external debug tool sends debug CAN messages to one of the MCAN modules of the MCU. The debug messages have specific identifiers and have to be sent/received in a specific order but may be interleaved with non-debug CAN messages. A mechanism internal to the MCAN module consisting of debug message filtering and a debug message state machine handles the incoming CAN debug messages, stores them correctly in a specified RX Buffer and triggers DMA data transfers between MCAN and JTAGM.

The JTAGM acts as a JTAG master within the device and is used to generate internal JTAG messages. Three CAN messages are necessary to generate the JTAG messages.

The following resources need to be initialized in the application software to use the 'Debug over CAN' feature:

- Three MCAN message filters need to be configured for the debug messages
- The DMA channel mux needs to be enabled for MCAN triggering
- DMA channel(s) need to be programmed for message transfer between MCAN and JTAGM, and to trigger the MCAN transmission to send the debug data back to the tool

1.1 MCAN

The Modular CAN (MCAN) modules are part of the Controller Area Network (CAN) subsystem alongside the Time Triggered CAN (TTCAN) modules and the CAN RAM controller. The MCAN performs communication according to the CAN protocol specification 2.0 part A, B and to CAN FD 1.0.

All functions concerning the handling of messages are implemented by the RX Handler and the TX Handler. The RX Handler manages message acceptance filtering, the transfer of received messages from the CAN Core to the Message RAM as well as providing receive message status information. The TX Handler is responsible for the transfer of transmit messages from the Message RAM to the CAN Core as well as providing transmit status information. Acceptance filtering is implemented by a combination of up to 128 standard filter elements or 64 extended filter elements where each one can be configured as a range, as a bit mask, or as a dedicated ID filter.

MCAN modules _1 and _2 can be used for the ‘Debug over CAN’ application. Its role in this application is to receive and handle the incoming debug messages from an external tool, trigger the DMA transfer sequence, and to send internal debug messages back to the external tool.

Incoming messages are identified as debug messages through a unique CAN message identifier and then stored in a specified location in the receive buffer. After the three CAN debug messages have been received correctly and stored in the debug receive buffer, the MCAN triggers a DMA transfer to send the data to the JTAGM. The MCAN also receives data back from the JTAGM via the DMA. This data is to be sent back to external tool. The CAN reception and transmission of these debug messages has to be configured by the user.

1.2 eDMA

The Enhanced Direct Memory Access (eDMA) module is a highly-programmable data-transfer engine optimized to minimize the intervention from the host processor. It is intended for use in applications where the data size to be transferred is statically known and not defined within the data packet itself.

The eDMA is used to transfer the information between the MCAN module, that sends and receives the debug messages to and from an external tool, and the JTAGM, that creates the internal JTAG traffic.

Six DMA transfers are required to accomplish one full cycle of the ‘Debug over CAN’ sequence. A sequence of DMA transfers is triggered by one of the MCAN modules in response to receiving debug messages from an external tool. A DMA Mux has to be configured to route the MCAN trigger to a DMA channel, and the DMA has to be enabled for MCAN triggering.

Any of the available DMA channels can be used and it is up to the user to set up Transfer Control Descriptor (TCD). Only the first of the sequence of five DMA transfers is triggered by the MCAN. The subsequent triggers have to be generated from the DMA itself. Both channel linking or the scatter/gather mechanism may be used. It is recommended to use the more resourceful scatter/gather mechanism. However, for ease of understanding, channel linking is the method of choice for the example in this application note. For more details on scatter/gather and channel linking please refer to the device’s reference manual and/or related application notes.

1.3 JTAGM

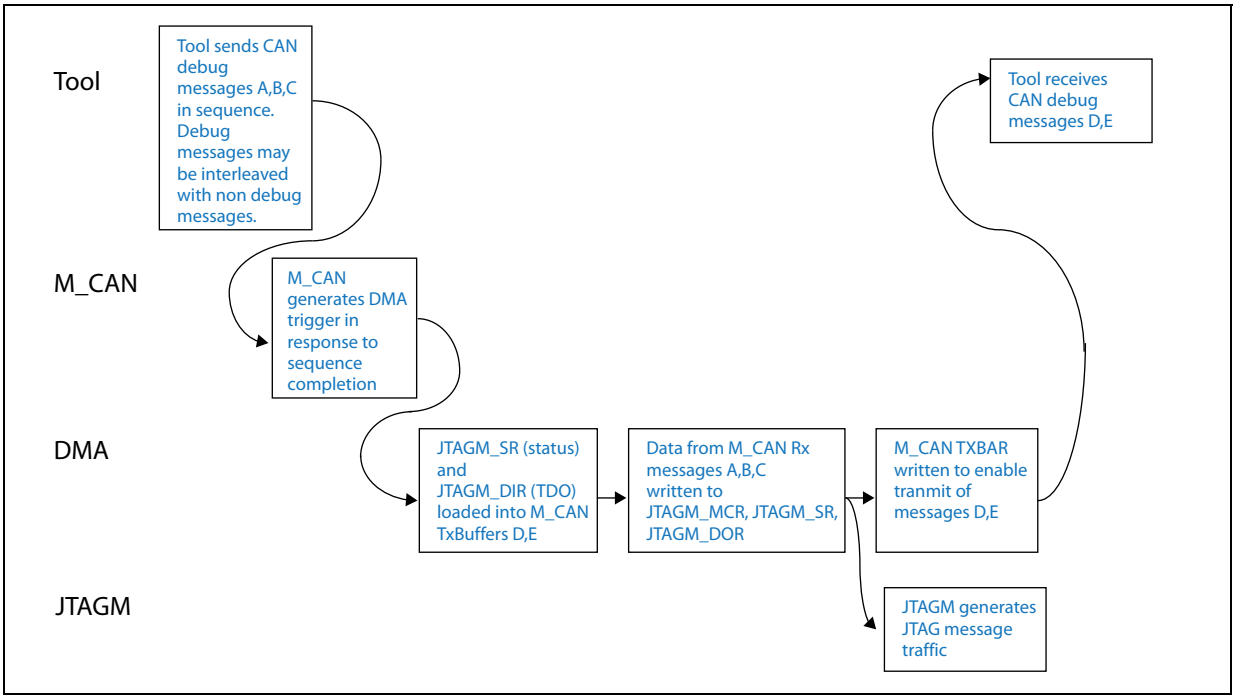
The JTAG Master (JTAGM) is a module that is able to act as JTAG master inside the device. The module has a parallel interface that can exchange data with the debug LFAST module and also a register interface that can be accessed via customer software or in the case of Debug over CAN, via eDMA. Data transferred to this module is used to produce TCK, TMS,

TDI and TRST outputs and to accept TDO inputs. The JTAGM is connected in the device to allow these five signals to connect to the JTAG controller (JTAGC) as if the JTAG data is coming from an outside tool. The JTAGM generates all required JTAG scan chains to allow software and high speed serial communication access to all JTAG mapped resources.

1.4 Sequence description

Figure 2 gives a detailed overview of the Debug over CAN sequence.

Figure 2. Debug over CAN sequence - detailed



Typically debugging over CAN takes several individual ‘Debug over CAN’ cycles to accomplish a given task, e.g. a Nexus Read/Write Access. In one ‘Debug over CAN’ cycle an external tool sends three CAN debug messages A, B and C sequentially to the MCAN of the microcontroller. The debug messages may be interleaved with none debug messages but must be in the correct order. The CAN debug messages are identified by a unique CAN message identifier. When a debug message has been identified it is stored in the user defined RX Buffer in the CAN message RAM.

After the correct reception of debug messages A, B and C the MCAN triggers a DMA transfer. First, the content of the JTAGM status register (JTAGM_SR) and the JTAGM DATA IN 0 (JTAGM_DIR0) and JTAGM DATA IN 1 (JTAGM_DIR1) registers are transferred to the user defined MCAN TX buffer in the CAN message RAM. JTAGM_DIR0 and _DIR1 hold the debug data from the previous ‘Debug over CAN’ cycle and hence needs to be read back before starting a JTAG traffic generation sequence.

After reading back the data from the JTAGM Status and Data In registers, the relevant data from the debug messages A, B and C in the RX Buffer of the CAN message RAM is transferred to the JTAGM’s module control register (MCR), status register (SR) and data out registers (DOR0-3).

Writing the LSB in DOR3 sets the JTAGM_DOR3_SEND bit and internal JTAG traffic is generated depending on the data written to the JTAGM registers.

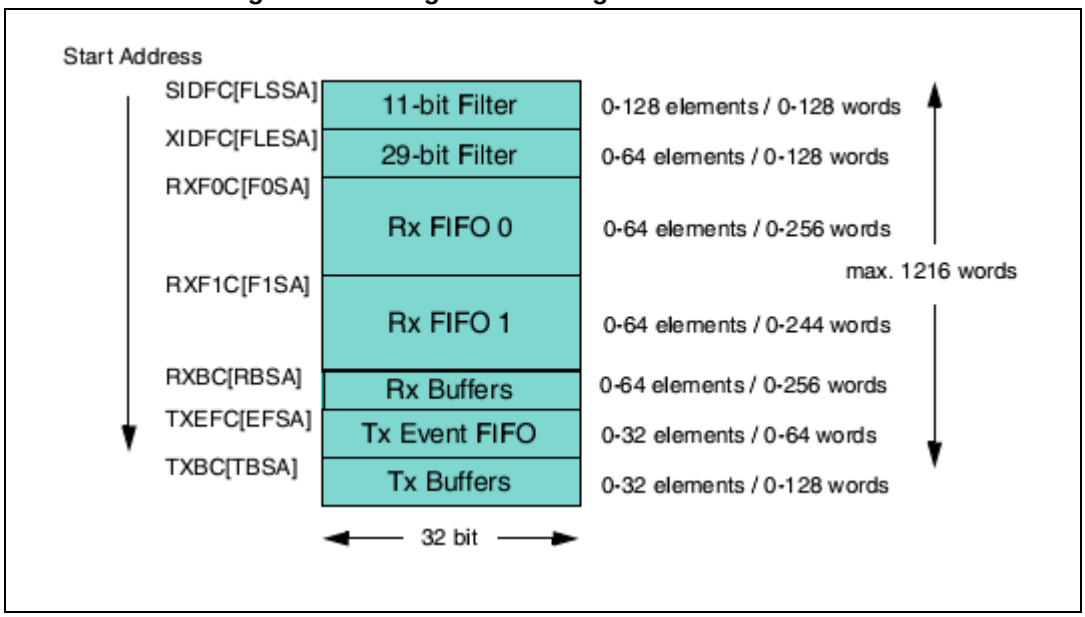
Parallel to that the MCAN sends the data that has just been stored in TX Buffers back to the external as debug messages D and E. The MCAN transmission is initiated using a DMA transfer to write transmission start bit of the given TX element.

Before starting the next ‘Debug over CAN’ cycle the external tool waits for the two CAN messages D and E coming back from the MCU’s MCAN. This provides enough time for the internal JTAG traffic to be finished and the data in JTAGM_DIR0 and _DIR1 has been updated.

2 MCAN debug message handling

Generally, the CAN modules on the SPC57xx/SPC58xx devices share 16 kByte of common CAN message RAM space. Within this CAN message RAM the user defines an individual area for each of the CAN modules. This area contains the message filters, Rx FIFO blocks, Rx Buffers, a Tx Event FIFO block and Tx Buffers as illustrated in the*Figure 3*.

Figure 3. Message RAM configuration for SPC57xx



The CAN modules accept CAN messages with matching ID. Any CAN message that is accepted by the CAN module will be stored as an element in either Rx FIFO_0, Rx FIFO_1 or Rx Buffer depending on the message filter that is configured for a given message ID. For debug messages the MCAN Rx Elements always follow the 16byte structure shown in *Figure 4*.

Figure 4. MCAN RX FIFO Element

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R0	ESI	XTD	RTR	ID[28:0]																												
R1	ANMF	FIDX[6:0]							res	EDL	BRS	DCL[3:0]				RXTS[15:0]																
R2	DB3[7:0]								DB2[7:0]								DB1[7:0]								DB0[7:0]							
R3	DB7[7:0]								DB6[7:0]								DB5[7:0]								DB4[7:0]							

The 8 data bytes DB0 to DB7 contain the data that will be transferred to the JTAGM by the DMA. If an extended filter is used the XTD bit is set to ‘1’ and all 29 bits of the ID field are used. For a standard ID bit [28:18] are used for the 11 bit long ID.

In order for the MCAN module to receive/accept messages and handle debug messages as desired the correct filter settings have to be programmed into the filter blocks of the CAN Message RAM Configuration area. A standard filter element is illustrated in *Figure 5* and an extended message filter element is illustrated in *Figure 6*.

Figure 5. Standard Message ID Filter Element

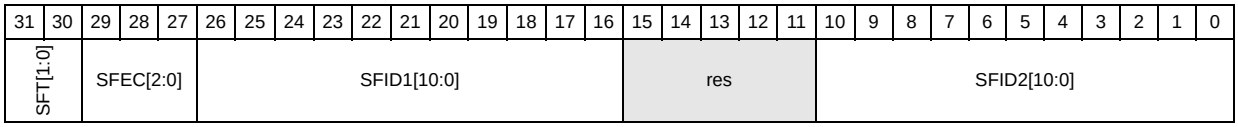
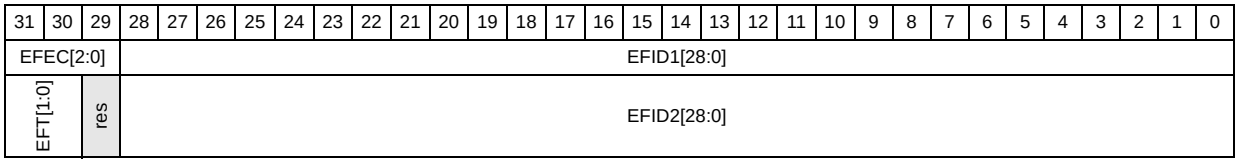


Figure 6. Extended Message ID Filter Element



SFID1/EFID1 is a unique identifier that matches the ID of the received CAN messages. The settings of SFEC/EFEC and SFID2/EFID2 determine how the received message will be handled, i.e. where it will be stored, if it is a debug message and so on.

The setting of SFEC/EFEC determines in which block of the configure message RAM the received CAN message is to be stored. For debug messages the SFEC/EFEC must be set to '111', and then the message is stored in the Rx Buffer. The setting of SFT/EFT will be ignored in this case and will not be discussed in this application note. The bits SFID2[10:9]/EFID2[10:9] decide whether the message will be treated as message A, B or C of the debug message sequence with the following key:

- 01 Debug Message A
- 10 Debug Message B
- 11 Debug Message C

SFID2[5:0]/ EFID2[5:0] defines the offset to the Rx Buffer Start Address RXBC.RBSA for storage of a matching message.

Table 2 gives an overview of typical settings:

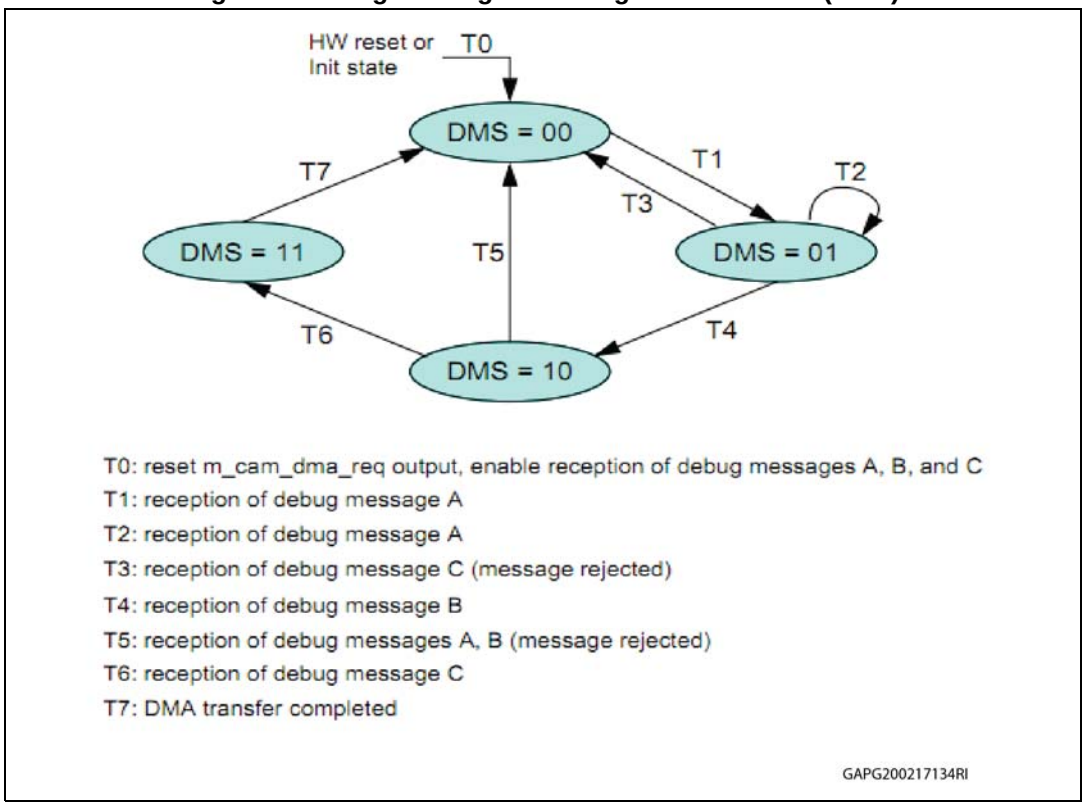
Table 2. Debug Message Filter Definition

Debug Message	SFID1[10:0] EFID1[28:0]	SFID2[10:9] EFID2[10:9]	SFID2[5:0] EFID2[5:0]
Message A, Rx FIFO 1 element 61	ID debug message A	01	00 0000
Message B, Rx FIFO 1 element 62	ID debug message B	10	00 0001
Message C, Rx FIFO 1 element 63	ID debug message C	11	00 0010

SFID2/EFID2[8:6] are used to control the filter event pins at the Extension Interface. For more details please refer to the device's reference manual.

The debug messages A, B and C may be interleaved with other CAN messages but must be received in the correct order. The so called Debug Message Handling State Machine controls that the correct order of debug messages is received, and the status of the status machine is indicated by the DMS bit field in the JTAGM status register (JTAGM_SR[DMS]). Debug messages that arrive in the wrong order are rejected, and the state machine is reset to its initial state (JTAGM_SR[DMS] = 00).*Figure 7*shows a transition diagram of the DMS.

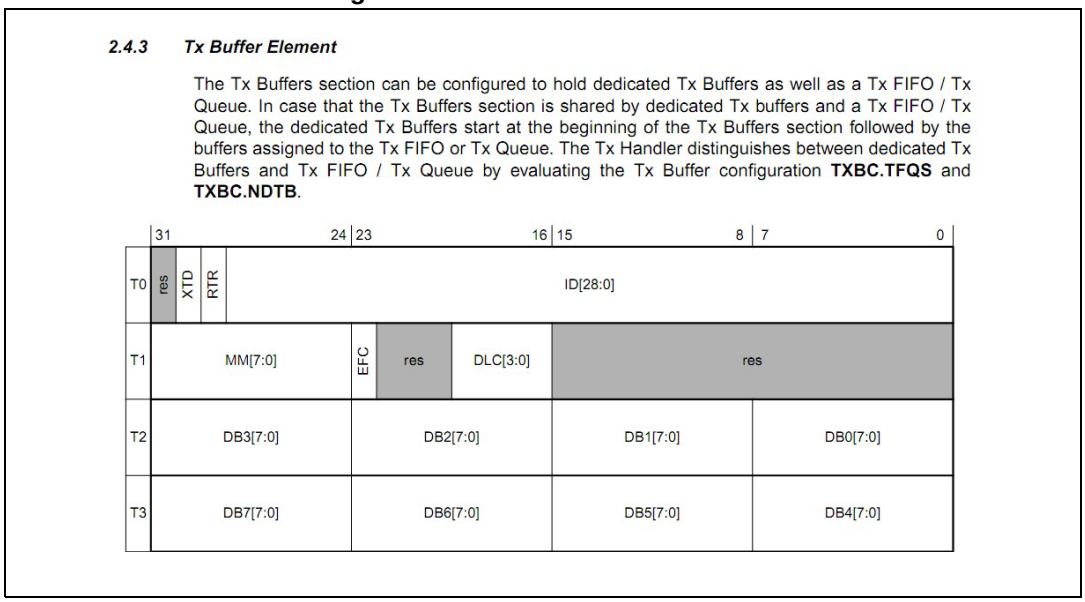
Figure 7. Debug Message Handling State Machine (DMS)



The correct reception of debug message C triggers a DMA transfer (if configured correctly). However, before the debug data is transferred to the JTAGM, the status and the debug data from the previous ‘Debug over CAN’ cycle must be read back from the JTAGM and transferred to the MCAN to be sent to the external tool.

Figure 8 illustrates the structure of the 16byte MCAN Tx Buffer element recommended to use for ‘Debug over CAN’.

Figure 8. MCAN TX Buffer Element



2.4.3 Tx Buffer Element

The Tx Buffers section can be configured to hold dedicated Tx Buffers as well as a Tx FIFO / Tx Queue. In case that the Tx Buffers section is shared by dedicated Tx buffers and a Tx FIFO / Tx Queue, the dedicated Tx Buffers start at the beginning of the Tx Buffers section followed by the buffers assigned to the Tx FIFO or Tx Queue. The Tx Handler distinguishes between dedicated Tx Buffers and Tx FIFO / Tx Queue by evaluating the Tx Buffer configuration **TXBC.TFQS** and **TXBC.NDTB**.

	31	24	23	16	15	8	7	0
T0	res	XTD	RTR	ID[28:0]				
T1	MM[7:0]		EFC	res	DLC[3:0]	res		
T2	DB3[7:0]		DB2[7:0]		DB1[7:0]		DB0[7:0]	
T3	DB7[7:0]		DB6[7:0]		DB5[7:0]		DB4[7:0]	

The lower 8byte contain the debug data. The upper 8 byte have to be configured so the external tool receiving the CAN message will successfully accept the CAN message.

Two dedicated Tx Buffer elements should be used for the two CAN messages that are send back to the external CAN debug tool. The data bytes of these Tx Buffer elements are written by the DMA with the data from the JTAGM. It is recommended to initialize the 8 byte of message overhead during the initialization of the ‘Debug over CAN’ feature.

3 DMA Transfers between MCAN and JTAGM

The DMA transfer sequence is triggered by the MCAN when debug message C has been received successfully. A total of six DMA transfers is required to complete one sequence of debug message transfer between MCAN and JTAGM.

- First, two transfers are required to send data from the JTAGM to the transmit buffers of the MCAN
- Thereafter, three transfers are required to send the new debug messages from the MCAN debug message RX buffer to the JTAGM
- One transfer is required to trigger the MCAN transmission to send data back to the external CAN debug tool

Note: One additional transfer is required when using the 'Debug over CAN' feature with SPC5744K cut1. Due to an errata the new data flags in the MCAN_NDAT registers have to be reset by a DMA transfer at the end of each 'Debug over CAN' cycle. It is recommended to do that before triggering the MCAN transmission with the final DMA transfer.

3.1 Enabling the eDMA trigger from the MCAN

The modules MCAN_1 and MCAN_2 have the DMA trigger mechanism implemented. The DMA Mux must be configured to route MCAN to one of the 16 DMA channels. The *Table 3* lists the DMA Mux and source numbers for the two MCAN modules:

Table 3. DMA Mux - MCAN Allocation

DMA Mux instance	Source	MCAN instance
DMAMUX_0	12	MCAN_1
DMAMUX_0	13	MCAN_2
DMAMUX_5(1)	34	MCAN_1
DMAMUX_5(1)	35	MCAN_2

1. Not available on SPC574K72.

The DMA Mux instance is configured in the DMACHMUX_CHCONFIG register. The channel must be enabled and the source of the trigger needs to be selected: The MSB in this 8bit register enable the channel and the six LSBs select the source.

Furthermore, the DMA itself needs to be configured to enable triggering from the same afore-configured DMA Mux channel.

Finally, the Transfer Control Descriptors (TCD) have to be configured. In the TCDs the user defines properties for the DMA transfers like source and destination address and size, channel linking options, and more. Please refer to the example section or the device reference manual for more details.

In order to make the sequence of six DMA transfers happen it is recommended to use channel linking or scatter-gather mechanism.

3.2 DMA Transfers: JTAGM to MCAN

The JTAGM status register (SR) and the data in registers (DIR0-1, containing the JTAG TDO data) hold the debug data from the previous messaging cycle and have to be transferred to the MCAN TX buffers.

The following three register contents need to be transferred to the MCAN TX Buffer to form CAN debug messages D and E that will be send back to the external CAN debug tool to complete the 'Debug over CAN' cycle:

1. Read JTAGM_SR -> Write to DB[3:0] of M_CAN Tx element configured for ID D
2. Read JTAGM_DIR0 -> Write to DB[7:4] of M_CAN Tx element configured for ID D
3. Read JTAGM_DIR1 -> Write to DB[3:0] of M_CAN Tx element configured for ID E

3.3 MCAN to JTAGM

The data from the three CAN debug messages has to be transferred to six JTAGM registers to generate the internal JTAG traffic. The following list defines which data portions from the CAN debug messages are transferred to the individual JTAGM registers:

1. Read DB[3:0] of M_CAN Rx element configured for ID A -> Write to JTAGM_MCR
2. Read DB[7:4] of M_CAN Rx element configured for ID A -> Write to JTAGM_SR
3. Read DB[3:0] of M_CAN Rx element configured for ID B -> Write to JTAGM_DOR0
4. Read DB[7:4] of M_CAN Rx element configured for ID B -> Write to JTAGM_DOR1
5. Read DB[3:0] of M_CAN Rx element configured for ID C -> Write to JTAGM_DOR2
6. Read DB[7:4] of M_CAN Rx element configured for ID C -> Write to JTAGM_DOR3

3.4 MCAN TX Trigger

The final DMA transfer is used to trigger the MCAN transmission to send data back to the tool.

The MCAN transmission is triggered by writing the transmission start bit of the dedicated Tx Buffer elements in the MCAN_TXBAR register. Using the DMA this can be done by configuring the address of a constant value as the source address in the TCD, and the MCAN_TXBAR register as its destination address.

4 JTAG Traffic Generation

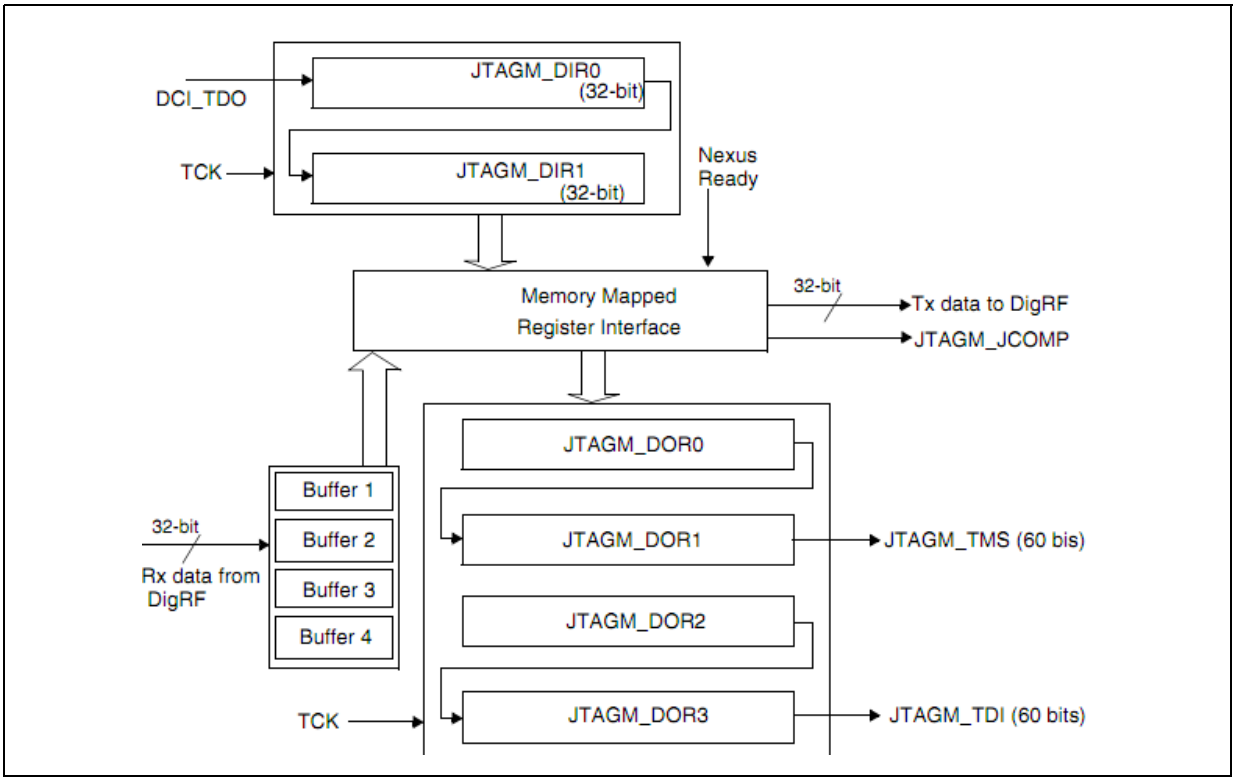
The JTAGM master is used to generate internal JTAG traffic by programming the JTAGM registers to interface with JTAG controller (JTAGC) and the Debug and Calibration Interface (DCI).

4.1 JTAG Master (JTAGM)

The JTAG Master (JTAGM) is a module that is able to act as JTAG master inside the device. The module has a parallel interface that can exchange data with another serial communication module (such as the LFAST module) or via customer software writing to the JTAGM registers.

Figure 9 provides a block diagram for the JTAGM.

Figure 9. JTAGM block diagram



The JTAGM has the capability to differentiate between data from software and LFAST. It then sends this data on TDI, TMS and TCK to the Debug and Calibration Interface (DCI). The JTAGM also receives serial data through TDO from the DCI and transfers this data to the software accessible registers or the LFAST through another parallel interface.

The JTAGM has the following registers:

- Configuration register (MCR)
- Status register (SR)
- Four data output registers (DOR0 – 3)
- Two data input registers (DIR0 – 1)

To generate JTAG signals by software the MCR and the DOR0 – 3 have to be written:

– **TCK** is generated from the device system clock and may be divided depending on the configuration of TCKSEL in JTAGM_MCR:

Figure 10. JTAGM block diagram

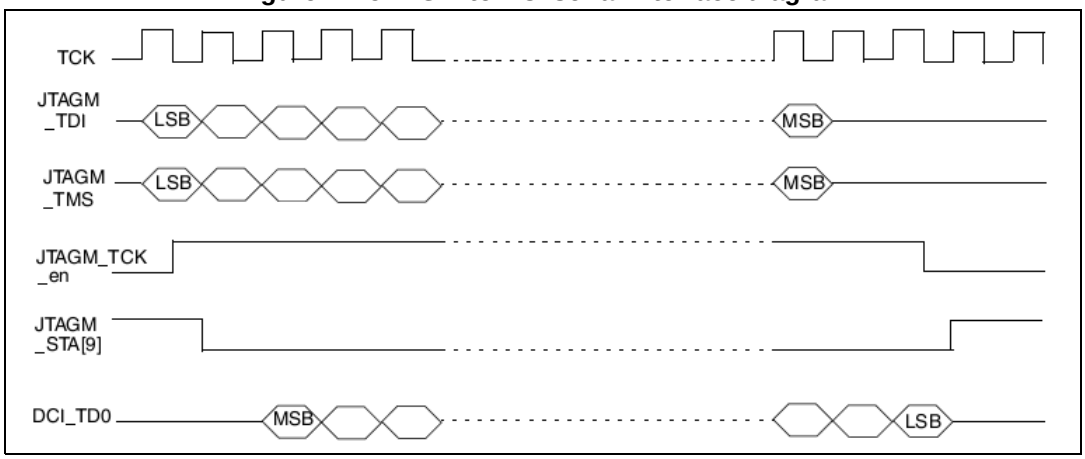
27:29 TCKSEL	TCK clock frequency selection. 000 TCK same as system clock 001 TCK is system clock ÷ 2 010 TCK is system clock ÷ 3 ... 111 TCK is system clock ÷ 8

Furthermore the MCR bits jtag_JCOMP, DTM and evti0_assert need to be written to ‘1’.

Data is pushed to the modules via a 32-bit wide parallel interface. The data is in the form of 60 bits of **TMS** data and 60 bits of **TDI** data. This data is the logical state to be driven onto the TMS and TDI output pins on each of the 60 TCK cycles. During these 60 TCK cycles, the TDO pin is sampled and the 60 bits of sampled data is transferred to the 32-bit wide module interface. This concept allows complete flexible generation on TMS and TDI data and capture of TDO data. The only limitation is that JTAG signals can only be generated in 60 TCK cycles packets. Extra cycles are wasted in idle cycles at the end of a scan chain.

The TMS signals are generated by the 60 MSBs of JTAGM_DOR0 + _DOR1. The TDI signals are generated by the 60 MSBs of JTAGM_DOR2 + _DOR3.
The LSB in JTAGM_DOR3 is the ‘Send’ bit. Writing ‘1’ to it will start the JTAG generation.
Figure 11 shows a diagram of the signals between the JTAGM and the DCI.

Figure 11. JTAGM to DCI serial interface diagram

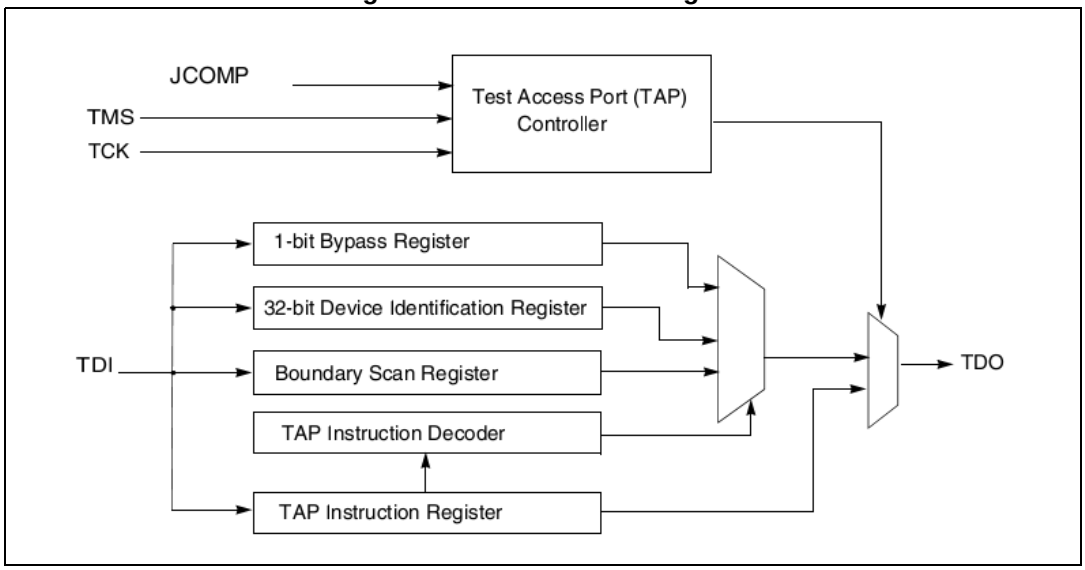


Bit ‘Idle’ in JTAGM_SR indicates if JTAG traffic is ongoing or finished.

4.2 JTAG Controller (JTAGC)

The JTAGC block provides the means to test chip functionality and connectivity while remaining transparent to system logic when not in test mode. Testing is performed via a boundary scan technique, as defined in the IEEE 1149.1-2001 standard. All data input to and output from the JTAGC block is communicated in serial format. *Figure 12* shows the block of the JTAGC.

Figure 12. JTAGC block diagram



The JTAGC consists of five signals that connect to off-chip and on-chip development tools and allow access to test support functions:

- Test Clock Input (TCK) is an input pin used to synchronize the test logic and control register access through the TAP.
- Test Data Input (TDI) is an input pin that receives serial test instructions and data. TDI is sampled on the rising edge of TCK
- Test Data Output (TDO) is an output pin that transmits serial output for test instructions and data. TDO is three-stateable and is actively driven only in the Shift-IR and Shift-DR states of the TAP controller state machine
- Test Mode Select (TMS) is an input pin used to sequence the IEEE 1149.1-2001 test control state machine. TMS is sampled on the rising edge of TCK.
- The JCOMP signal provides IEEE 1149.1-2001 compatibility and provides the ability to share the TAP. The JTAGC TAP controller is enabled when JCOMP is set to the JTAGC enable encoding, otherwise the JTAGC TAP controller remains in reset.

4.2.1 JTAGC Instruction Register

The JTAGC block registers are not memory mapped and only accessible through the TAP interface, including data registers and the instruction register.

The JTAGC block uses a 6-bit instruction register as shown in *Figure 13*.

Figure 13. 6-bit Instruction Register

	5	4	3	2	1	0
R	0	0	0	0	0	1
W	Instruction Code					
Reset:	0	0	0	0	0	1

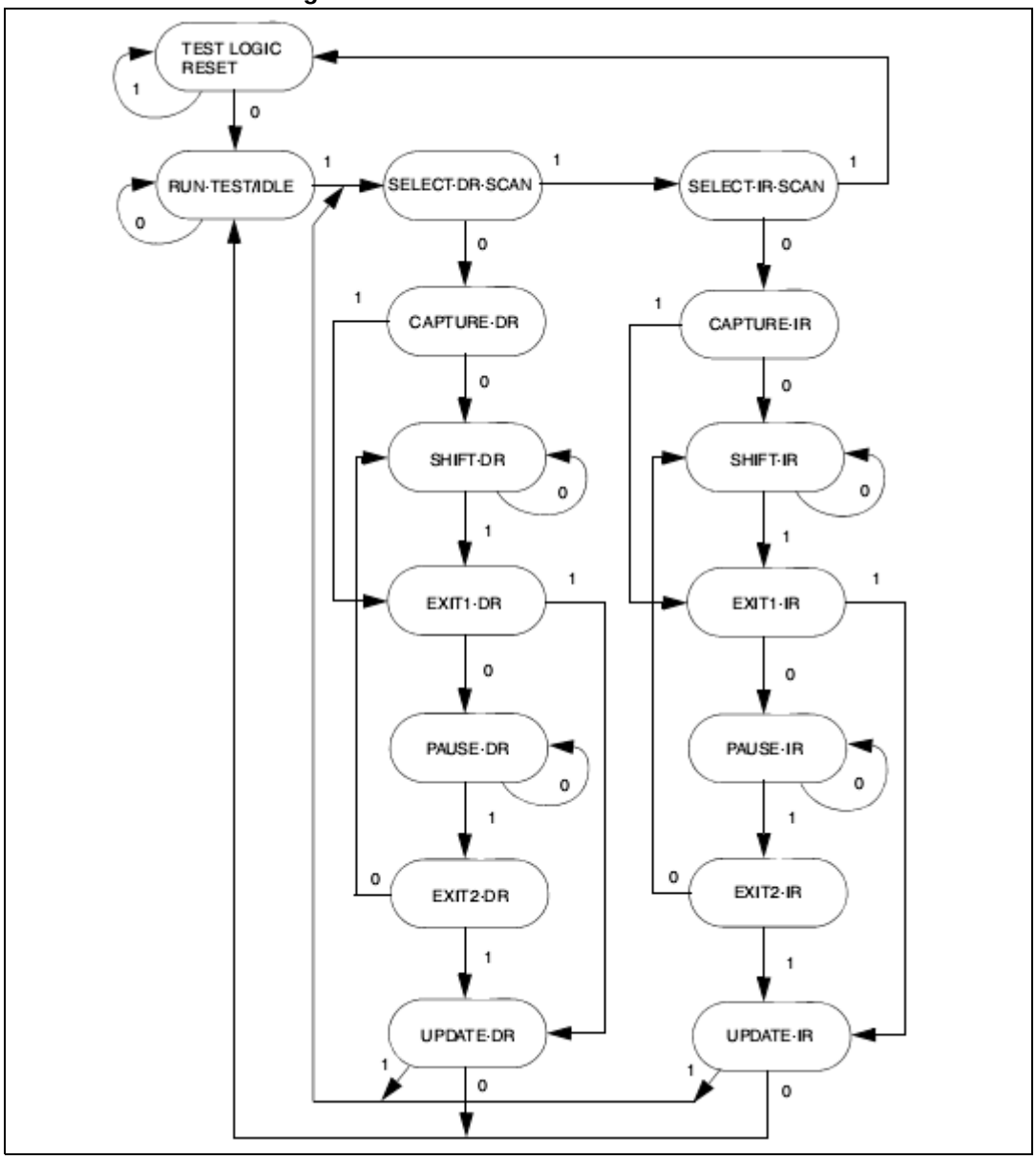
The instruction register allows instructions to be loaded into the block to select the test to be performed or the test data register to be accessed or both. Instructions are shifted in through TDI while the TAP controller is in the Shift-IR state, and latched on the falling edge of TCK in the Update-IR state. The latched instruction value can only be changed in the Update-IR and Test-Logic-Reset TAP controller states. Synchronous entry into the Test-Logic-Reset state results in the IDCODE instruction being loaded on the falling edge of TCK. Asynchronous entry into the Test-Logic-Reset state results in asynchronous loading of the IDCODE instruction. During the Capture-IR TAP controller state, the instruction shift register is loaded with the value 000001b, making this value the register's read value when the TAP controller is sequenced into the Shift-IR state.

For information on the other JTAGC registers please refer to the reference manual.

4.2.2 JTAGC TAP Controller

The TAP controller is a synchronous state machine that interprets the sequence of logical values on the TMS pin.*Figure 14* shows the machine's states. The value shown next to each state is the value of the TMS signal sampled on the rising edge of the TCK signal.

Figure 14. TAP controller state machine



To initialize the JTAGC block and enable access to registers, the following sequence is required:

1. Set the JCOMP signal to the JTAGC enable value, thereby enabling the JTAGC TAP controller
2. Load the appropriate instruction for the test or action to be performed

Access to the JTAGC data registers is achieved by loading the instruction register with any of the JTAGC block instructions while the JTAGC is enabled. Instructions are shifted in via the Select-IR- Scan path and loaded in the Update-IR state. At this point, all data register access is performed via the Select-DR-Scan path.

The Select-DR-Scan path is used to read or write the register data by shifting the data in and out (LSB first) during the Shift-DR state. When reading a register, the register value is loaded into the IEEE 1149.1-2001 shifter during the Capture-DR state. When writing a register, the

value is loaded from the IEEE 1149.1-2001 shifter to the register during the Update-DR state. When reading a register, there is no requirement to shift out the entire register contents. Shifting may be terminated once the required number of bits has been acquired.

The JTAGC block implements the IEEE 1149.1-2001 defined instructions listed *Table 4*.

This section gives an overview of each instruction; refer to the IEEE 1149.1-2001 standard for more details. All undefined opcodes are reserved.

Table 4. 6-bit JTAG Instructions for SPC57xx

Instruction	Code[5:0]	Instruction Summary
IDCODE	000001	Selects device identification register for shift.
SAMPLE/PRELOAD	000010	Selects boundary scan register for shifting, sampling and preloading without disturbing functional operation.
SAMPLE	000011	Selects boundary scan register for shifting and sampling without disturbing functional operation.
EXTEST	000100	Selects boundary scan register while applying preloaded values to output pins and asserting functional reset.
Factory debug reserved	000101	Intended for factory debug only.
Factory debug reserved	000110	Intended for factory debug only.
ENABLED_JTAG_PASSWORD	000111	Selects JTAG_PASSWORD register.
HIGHZ	001001	Selects bypass register while tree-stating all output pins and asserting functional reset.
Factory debug reserved	001010	Intended for factory debug only.
CLAMP	001100	Selects bypass register while applying preloaded values to output pins and asserting functional reset.
ENABLED_DCI_CR	001110	Enables access to DCI CR register.
ENABLED_DCI_PINCR	001111	Enables access to DCI PINCR register.
ACCESS_AUX_x	100000-111110	Grants one the auxiliary TAP controllers ownership of the TAP as shown in the cells below.
Reserved	100000	Reserved
ACCESS_AUX_NR_PD	100001	Enables access to the NAR TAP controller on PD.
Reserved	100010	Reserved
Reserved	100011	Reserved
Reserved	100100	Reserved
Factory test reserved	100101	Intended for factory test only.
ACCESS_AUX_JDC	100110	Enables access to the JDC TAP controller to access the HSM challenge and response registers.
ACCESS_AUX_HSM	100111	Enables access to the HSM core TAP controller.
ACCESS_AUX_CORE_0	101000	Enables access to E200 Core 0
ACCESS_AUX_CORE_1	101001	Enables access to E200 Core 1
ACCESS_AUX_CORE_2	101010	Enables access to E200 Core 2

Table 4. 6-bit JTAG Instructions for SPC57xx (continued)

Instruction	Code[5:0]	Instruction Summary
Reserved	101011	Reserved
Reserved	101100	Reserved
Reserved	101101	Reserved
Reserved	101110	Reserved
Reserved	101111	Reserved
Reserved	110000	Reserved
Reserved	110001	Reserved
Reserved	110010	Reserved
ACCESS_AUX_GTM	110011	Enables access to GTM TAP controller.
ACCESS_AUX_BUS_MON_0	110100	Enables access to NXMC Client 0 TAP controller.
ACCESS_AUX_BUS_MON_1	110101	Enables access to NXMC Client 1 TAP controller.
Reserved	110110	Reserved
Reserved	110111	Reserved
Reserved	111000	Reserved
Reserved	111001	Reserved
ACCESS_AUX_SPU_A	111010	Enables access to SPU Client TAP controller.
Reserved	111011	Reserved
ACCESS_AUX_PARALLEL(1)	111100	Enables access to all cores in parallel.
Reserved	111101	Reserved
Reserved	111110	Reserved
BYPASS	111111	Selects bypass register for data operations.
Reserved(2)	All other opcodes	Decoded to select bypass register.

1. The use of this instruction is limited to the synchronous exit of debug mode via the execution of the GO+EXITOnCE command. The execution of the other instructions or commands is not supported.
2. The manufacturer reserves the right to change the decoding of reserved instruction codes in the future.

For a detailed description of the JTAGC please refer to the device's reference manual.

5 Example: Reading the JTAG ID: using SPC57xx

This example is a good starting point to detail how ‘Debug over CAN’ works. A complete Greenhills project containing all the code is provided with this application note.

The device identification (JTAG ID) register allows the revision number, part number, manufacturer, and design center responsible for the design of the part to be determined through the TAP. The device identification register is selected for serial data transfer between TDI and TDO when the IDCODE instruction is active. Entry into the Capture-DR state while the device identification register is selected loads the IDCODE into the shift register to be shifted out on TDO in the Shift-DR state. No action occurs in the Update-DR state.

IDCODE selects the 32-bit device identification register as the shift path between TDI and TDO. This instruction allows interrogation of the MCU to determine its version number and other part identification data. IDCODE is the instruction placed into the instruction register when the JTAGC block is reset.

In order to use ‘Debug over CAN’ to read the JTAG ID the user needs to configure three CAN debug messages that are send from external tool to be received by either MCAN_1 or MCAN_2. A DMA has to be configured to transfer the data from the MCAN module to the correct registers in the JTAGM.

5.1 Example Set-up overview

For the example the two modules MCAN_1 and MCAN_2 are connected externally. MCAN_2 emulates an external tool that sends the three CAN debug messages to MCU. MCAN_1 is configured to handle the incoming CAN debug messages.

In the main loop the device gets initialized first:

- The clocks are set up to standard use case scenario in MC_MODE_INIT()
- The eDMA is initialized in DMA_Init():
  - Seven TCDs are configured for the seven transfers; channel linking is used to trigger all the subsequent transfers after the first one
  - DMA channel mux is configured to connect MCAN_1 to the eDMA
  - DMA is enabled to be triggered by the MCAN
- The MCANs are initialized in MCAN1_init() and MCAN2_init():
  - CAN baud rate is set to 500kbit/s for both modules
  - CAN message RAM is configured. MCAN_1 has the Rx Buffer RXBC configured to store the CAN debug messages.
  - Two dedicated Tx Buffer elements are configured at the beginning of the MCAN_1 Tx Buffer. The first eight bytes of these two buffer elements are initialized with a specific identifier for the external tool. The eight data bytes of the two buffer elements will be written by the DMA with the data from the JTAGM
  - The pins are configured for MCAN use (in this example the pins that are used on the evaluation board are configured)
- MCAN_ID_init() initializes the filters in the standard filter area of the configuration RAM for MCAN_1 and MCAN_2. The MCAN_2 filter is configured to accept the CAN messages that come back from the MCU. The MCAN_1 filters are configured to treat messages with matching IDs as debug messages and store them in the Rx Buffer

The software initialization used in this example can be re-used or edited for any application. However, for real applications two changes would probably be made:

1. Different/specific filter IDs would be used
2. DMA scatter/gather mechanism would be used instead of channel linking in order to reduce the number of used DMA channels

After completing the initialization the MCU is now able to receive and handle CAN debug messages through its MCAN_1 module. In order to keep the system integrated the debug messages are sent from the MCAN_2 module via the external connection to the MCAN_1 by function MCAN2_tx(). The MCAN2 then waits for two debug messages to be sent back from the MCAN_1. This sequence is repeated in this example in order to receive the JTAG ID in the MCAN_2's receive buffers.

As it is not possible to run the example with a JTAG tool (debugger) attached, the LINFlexD module is used in UART mode to transmit the content of the RX buffer of MCAN_2 back to a hyperterminal. Transmitted are the lower eight bytes of the RX Buffer element that contain the TDO data from the JTAGM.

The JTAG ID is device Dependant. During the development of the project a MCP5744K device was used, and the following values had been transmitted from the MCAN_2 RX Buffer to the Hyperterminal:

– 0x41060198
– 0x10a80703

Those values contain the JTAG ID which can be found after ordering the bytes and bits correctly: First, the LINFlexD inverts the order of the bytes which gives:

– 0x98010641 and 0x0307a810 (in hex) or
– 1001.1000.0000.0001_0000.0110.0100.0001 and 0000.0011.0000.0111_1010.1000.0001.0000

Second, the JTAGC shifts the bits out by LSB first, hence the whole block has to be inverted bitwise:

– 0000.1000.000**1**.0**1**0**1**_**1110.0000.1100.0000** -
– **1000.001**0.0110.0000_1000.0000.0001.1001

This does contain the JTAG ID as highlighted in bold. The JTAG ID is: 0x0AF06041.

6 Example: Reading the JTAG ID using SPC58xx

This example is a good starting point to detail how ‘Debug over CAN’ works. A complete Greenhills project containing all the code is provided with this application note.

The device identification (JTAG ID) register allows the revision number, part number, manufacturer, and design center responsible for the design of the part to be determined through the TAP. The device identification register is selected for serial data transfer between TDI and TDO when the IDCODE instruction is active. Entry into the Capture-DR state while the device identification register is selected loads the IDCODE into the shift register to be shifted out on TDO in the Shift-DR state. No action occurs in the Update-DR state.

IDCODE selects the 32-bit device identification register as the shift path between TDI and TDO. This instruction allows interrogation of the MCU to determine its version number and other part identification data. IDCODE is the instruction placed into the instruction register when the JTAGC block is reset.

In order to use ‘Debug over CAN’ to read the JTAG ID the user needs to configure three CAN debug messages that are send from external tool to be received by either MCAN_1 or MCAN_2. A DMA has to be configured to transfer the data from the MCAN module to the correct registers in the JTAGM.

6.1 Example mainboard and minimodule configuration

The code has been developed on top of SPC57xxMB mainboard and SPC58XXADPT64S minimodule:

1. Connect pin 0 of P8 (PA[0]) to pin 1 of P7 (D2 user LED)
2. Connect pin 9 of P9 (PB[9]) to pin 4 of J32 (CAN2 RX)
3. Connect pin 10 of P9 (PB[10]) to pin 2 of J32 (CAN2 TX)
4. Connect pin 3 of P10 (PC[3]) to pin 2 of J14 (SCI RX)
5. Connect pin 4 of P10 (PC[4]) to pin 2 of J13 (SCI TX)
6. Connect pin 2 of J5 (CAN DB9 CANH) with pin 2 of J6
7. Connect pin 7 of J5 (CAN DB9 CANL) with pin 7 of J6
8. Connect J19 serial port to PC using 115200 baud, 8,N,1

6.2 Example set-up overview

For the example the two modules MCAN_1 and MCAN_2 are connected externally. MCAN_2 emulates an external tool that sends the three CAN debug messages to MCU. MCAN_1 is configured to handle the incoming CAN debug messages.

In the main loop the device gets initialized first:

- The clocks are set up to standard use case scenario in clock_init()
- The LINFlexD is set up to 115200 baud in UartInit()
- The eDMA is initialized in DMA_Init():
  - Seven TCDs are configured for the seven transfers; channel linking is used to trigger all the subsequent transfers after the first one
  - DMA channel mux is configured to connect MCAN_1 to the eDMA
  - DMA is enabled to be triggered by the MCAN
- The MCANs are initialized in CAN_init():
  - CAN baud rate is set to 1Mbit/s for both modules
  - CAN message RAM is configured. MCAN_1 has the Rx Buffer RXBC configured to store the CAN debug messages.
  - Two dedicated Tx Buffer elements are configured at the beginning of the MCAN_1 Tx Buffer. The first eight bytes of these two buffer elements are initialized with a specific identifier for the external tool. The eight data bytes of the two buffer elements will be written by the DMA with the data from the JTAGM
  - The pins are configured for MCAN use (in this example the pins that are used on the evaluation board are configured)
  - Initializes the filters in the standard filter area of the configuration RAM for MCAN_1 and MCAN_2. The MCAN_2 filter is configured to accept the CAN messages that come back from the MCU. The MCAN_1 filters are configured to treat messages with matching IDs as debug messages and store them in the Rx Buffer

6.3 Example: Reading the JTAG ID

The software initialization used in this example can be re-used or edited for any application. However, for real applications two changes would probably be made:

Different/specific filter IDs would be used

DMA scatter/gather mechanism would be used instead of channel linking in order to reduce the number of used DMA channels

After completing the initialization the MCU is now able to receive and handle CAN debug messages through its MCAN_1 module. In order to keep the system integrated the debug messages are sent from the MCAN_2 module via the external connection to the MCAN_1 by function MCAN2_SUB0_tx(). The MCAN2 then waits for two debug messages to be sent back from the MCAN_1. This sequence is repeated in this example in order to receive the JTAG ID in the MCAN_2's receive buffers.

As it is not possible to run the example with a JTAG tool (debugger) attached, the LINFlexD module is used in UART mode to transmit the content of the RX buffer of MCAN_2 back to a hyperterminal. Transmitted are the lower eight bytes of the RX Buffer element that contain the TDO data from the JTAGM.

The JTAG ID is device dependent. During the development of the project a SPC582B60E1 cut 2.1 device was used, and the following values had been transmitted from the MCAN_2 RX Buffer to the Hyperterminal:

- 0x98010641
- 0x00144410

Those values contain the JTAG ID which can be found after ordering the bytes and bits correctly:

- 0x98010641 and 0x00144410 (in hex) or
- 1001 1000 0000 0001 0000 0110 0100 0001 and
- 0000 0000 0001 0100 0100 0100 0001 0000

Second, the JTAGC shifts the bits out by LSB first, hence the whole block has to be inverted bitwise:

- 0000 1000 00**10 0010 0010 1000 0000 0000**
- **1000 001**0 0110 0000 1000 0000 0001 1001

This does contain the JTAG ID as highlighted in bold. The JTAG ID is: 0x11140041.

6.4 JTAGC block instructions for SPC58xx

The JTAGC block implements the IEEE 1149.1-2001 defined instructions listed in *Table 5*. This section gives an overview of each instruction; refer to the IEEE 1149.1-2001 standard for more details. All undefined opcodes are reserved.

Table 5. 6-bit JTAG instructions for SPC58xx

Instruction	Code[5:0]	Instruction Summary
IDCODE	000001	Selects device identification register for shift
SAMPLE/PRELOAD	000010	Selects boundary scan register for shifting, sampling, and preloading without disturbing functional operation
SAMPLE	000011	Selects boundary scan register for shifting and sampling without disturbing functional operation
EXTEST	000100	Selects boundary scan register while applying preloaded values to output pins and asserting functional reset
Reserved	000101	Reserved
Reserved	000110	Reserved
ENABLE_JTAG_PASSWORD	000111	Selects JTAG_PASSWORD register
HIGHZ	001001	Selects bypass register while three-stating all output pins and asserting functional reset
Reserved	001010	Reserved

Table 5. 6-bit JTAG instructions for SPC58xx (continued)

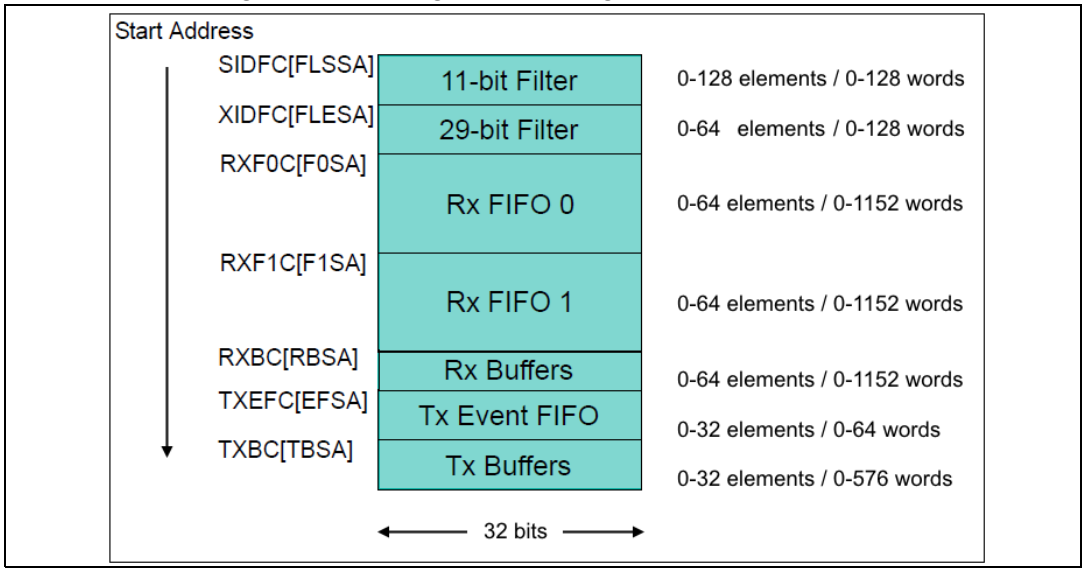
Instruction	Code[5:0]	Instruction Summary
CLAMP	001100	Selects bypass register while applying preloaded values to output pins and asserting functional reset
ENABLE_DCI_CR	001110	Enables access to DCI CR register
ACCESS_AUX_x	100000-111110	Grants one of the auxiliary TAP controllers ownership of the TAP as shown in the cells below
Reserved	100000	Reserved
ACCESS_AUX_NPC_PD	100001	Enables access to the NPC TAP controller on production device
Reserved	100010	Reserved
Reserved	100011	Reserved
Reserved	100100	Reserved
Reserved	100101	Reserved
ACCESS_AUX_JDC	100110	Enables access to the JDC TAP controller
Reserved		Reserved
Reserved		Reserved
Reserved		Reserved
ACCESS_AUX_CORE_2	101010	Enables access to E200 Core 2
Reserved	101011	Reserved
Reserved	101100	Reserved
Reserved	101101	Reserved
Reserved	101110	Reserved
Reserved	101111	Reserved
Reserved	110000	Reserved
Reserved	110001	Reserved
Reserved	110010	Reserved
Reserved	110011	Reserved
ACCESS_AUX_BUS_MON_0	110100	Enables access to NXMC Client 0 TAP controller
Reserved	110101	Reserved
Reserved	110110	Reserved
Reserved	110111	Reserved
Reserved	111000	Reserved
Reserved	111001	Reserved
Reserved	111001	Reserved
ACCESS_AUX_SPU_A	111010	Enables access to SPU Client TAP controller

Table 5. 6-bit JTAG instructions for SPC58xx (continued)

Instruction	Code[5:0]	Instruction Summary
Reserved	111011	Reserved
CCESS_AUX_PARALLEL(1)	111100	Enables access to all cores in parallel
Reserved	111101	Reserved
Reserved	111110	Reserved
BYPASS	111111	Selects bypass register for data operations
Reserved(2)	All other opcodes	Decoded to select bypass register

1. The use of this instruction is limited to the synchronous exit of debug mode via the execution of the GO+EXIT OnCE command. The execution of other instructions or commands is not supported
2. The manufacturer reserves the right to change the decoding of reserved instruction codes in the future.

Figure 15. Message RAM configuration for SPC58xx



Appendix A Further information

A.1 Terms and abbreviations

This document uses the following terms and abbreviations.

Table 6. Terms and abbreviations

Term	Meaning
CAN	Controller Area Network – A bus system used in automotive applications
DCI	Debug and Calibration Interface
DMA	Direct Memory Access
DMS	Debug Message State Machine – State machine handling debug message in the MCAN
JTAGM	JTAG Master
LFAST	LVDS Fast Asynchronous Serial Transmission
LSB	Least Significant Bit
LVDS	Low Voltage Differential Signal
MCAN	Modular CAN module
MCU	Microcontroller
TCD	Transfer Control Descriptor

Revision history

Table 7. Document revision history

Date	Revision	Changes
04-Sep-2013	1	Initial release.
17-Sep-2013	2	Updated Disclaimer.
06-Mar-2017	3	Added SPC58xx example and code.