TMS320C6x Interface to External Flash Memory

APPLICATION REPORT: PRELIMINARY

Kyle Castille

November 19, 1997

TEXAS INSTRUMENTS

IMPORTANT NOTICE

TRADEMARKS

TMS320C6x Interface to External Flash Memory

Abstract

Interfacing external Flash Memory to the TMS320C62xx is simple when compared to previous generations of TI DSPs due to the advanced External Memory Interface (EMIF), which provides a glueless interface to a variety of external memory devices.

This document will describe the:

☐ EMIF's control registers and Asynchronous Interface signals

☐ Flash functionality and performance considerations

☐ Full example using TI's TMS29LF800

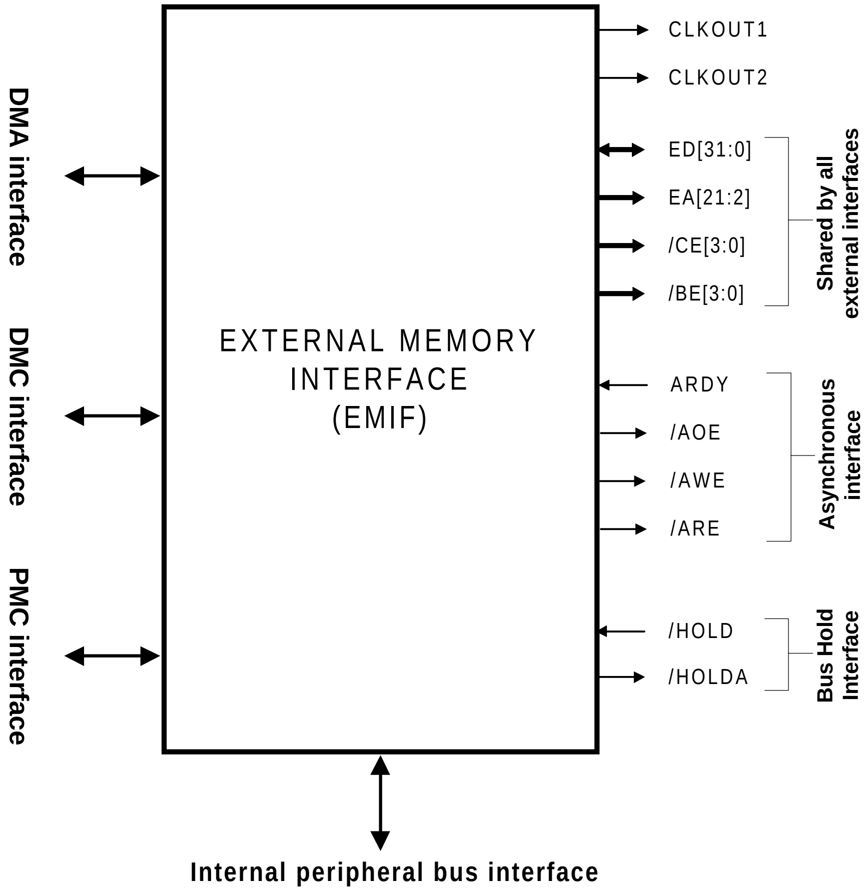
☐ Full example using TI's TMS29LF040

Overview of EMIF

EMIF Signal Descriptions

Figure 1 shows a block diagram of the EMIF. The EMIF is the interface between external memory and the other internal units of the ‘C6x. The interface with the processor is provided via the DMA controller, Program Memory Controller (PMC), and the Data Memory Controller (DMC)[1]. The signals described in Table 1, however, focus on the asynchronous interface and the shared interface signals.

Figure 1. Block Diagram of EMIF



---

[1] For a more detailed description of the interface between the EMIF and the other internal units of the C6x, see the TMS320C62xx Peripherals Reference Guide.

Table 1. EMIF Signal Descriptions : Shared Signals and Asynchronous Signals

Pin	(I/O/Z)	Description
CLKOUT1	O	Clock. Clock output - the CPU clock rate.
ED(31:0)	I/O/Z	Data I/O. 32-bit data input/output from external memories and peripherals.
EA(21:2)	O/Z	External Address output. Drives bits 21-2 of the byte address.
/CE0	O/Z	External /CE0 Chip Select. Active low chip select for CE space 0.
/CE1	O/Z	External /CE1 Chip Select. Active low chip select for CE space 1.
/CE2	O/Z	External /CE2 Chip Select. Active low chip select for CE space 2.
/CE3	O/Z	External /CE3 Chip Select. Active low chip select for CE space 3.
/BE(3:0)	O/Z	Byte Enables. Active low byte strobes. Individual bytes and halfwords can be selected for both read and write cycles. Decoded from 2 LSBs of the byte address.
ARDY	I	Ready. Asynchronous ready input used to insert wait states for slow memories and peripherals, such as Flash memory.
/AOE	O/Z	Output Enable. Active low output enable for asynchronous memory interface.
/AWE	O/Z	Write Strobe. Active low write strobe for asynchronous memory interface.
/ARE	O/Z	Read Strobe. Active low read strobe for asynchronous memory interface.
/HOLD	I	Active-low external bus hold (3-state) request.
/HOLDA	O	Active-low external bus hold acknowledge.

EMIF Registers

Control of the EMIF and the memory interfaces it supports is maintained through a set of memory mapped registers within the EMIF. A write to any EMIF register will not complete until all pending EMIF accesses which use that register have completed. The memory mapped registers are shown in Table 2.

Table 2. EMIF Memory Mapped Registers

Byte Address	Name
0x01800000	EMIF Global Control
0x01800004	EMIF CE1 Space Control
0x01800008	EMIF CE0 Space Control
0x0180000C	reserved
0x01800010	EMIF CE2 Space Control
0x01800014	EMIF CE3 Space Control

CE Space Control Registers

The four CE Space Control Registers (Figure 2) correspond to the four CE spaces supported by the EMIF. The MTYPE field identifies the memory type for the corresponding CE space. If MTYPE selects SDRAM or SBSRAM, the remaining fields in the register do not apply. If an asynchronous type is selected (32 bit asynchronous, or 8 or 16 bit ROM), the remaining fields specify the shaping of the address and control signals for access to that space. Table 3 contains a more detailed description of the asynchronous configuration fields. Modification of a CE Space Control Register does not occur until that CE space is inactive.

Figure 2. EMIF CE(0/1/2/3) Space Control Register Diagram

31 28	27 22	21 20	19 16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
RW, +1111	RW, +111111	RW, +11	RW, +1111

15 14	13 8	7	6 4	3 2	1 0
TA	READ STROBE	rsv	MTYPE	reserved	READ HOLD
RW, +11	RW, +111111	R, +0	RW, +010	R, +0	RW, +11

Table 3. EMIF CE(0/1/2/3) Space Control Registers Bitfield Description

Field	Description
READ SETUP WRITE SETUP	Setup width. Number of CLKOUT1 cycles of setup for address (EA) and byte enables (/BE(0-3)) before read strobe (/ASRE) or write strobe (/ASWE) falling. On the first access to a CE space this is also the setup after /CE falling.
READ STROBE WRITE STROBE	Strobe width. The width of read strobe (/ASRE) and write strobe (/ASWE) in CLKOUT1 cycles.
READ HOLD WRITE HOLD	Hold width. Number of CLKOUT1 cycles that address (EA) and byte strobes (/BE(0-3)) are held after read strobe (/ASRE) or write strobe (/ASWE) rising.

MTYPE	Memory Type.
	MTYPE=000b, 8-bit wide ROM
	MTYPE=001b, 16-bit wide ROM
	MTYPE=010b, 32-bit wide Asynchronous Interface
	MTYPE=011b, 32-bit wide SDRAM
	MTYPE=100b, 32-bit wide SBSRAM
	MTYPE=other, reserved

EMIF ROM Modes

The EMIF supports 8- and 16-bit wide ROM access modes as selected by the MTYPE field in the EMIF CE space contorl register. In reading data from these narrow-width memory spaces, the EMIF packs multiple reads into one 32-bit value. This mode is primarily intended for word access to 8bit and 16 bit ROM devices and operates as follows:

- Read operations always read 32 bits, regardless of the access size or the memory width.

- The address is shifted up appropriately to provide the correct address to the narrow memory. The shift amout is 1 for 16 bit ROM and 2 for 8 bit ROM. Thus the high address bits are shifted out and accesses wrap around if that CE space spacs the entier EA bus.

- The EMIF always reads the lower addresses first and packs these in the the LSBytes and packs subsequent acesses into the higher order bytes. Thus the expected packing format in ROM is always little-endian, regardless of the value of the LENDIAN bit.

Flash Memory Interface

The asynchronous interface offers users configurable memory cycle types, used to interface to a variety of memory and peripheral types; including SRAM, EPROM, FLASH, as well as FPGA and ASIC designs. This section, however, will focus on the interface between the EMIF and Flash memory, which is very similar to a ROM or EEPROM interface.

Table 3 shows that 8, 16, and 32 bit wide configurations are supported by the EMIF via its asynchronous interface. Flash memory is commonly available on the market in either 8 bit wide devices or configurable 8/16 bit wide devices. The configurable devices generally have 16 data I/O lines, but a mode select pin tells the device whether to operate in 8 or 16 bit mode.

Table 4 lists the EMIF's asynchronous interface pins and their mapping to pins on common Flash memory. Figure 3 shows an interface to 8/16 bit wide standard flash utilizing the 16 bit mode. In 8 bit mode, D[15] operates as the least significant bit of the address, thus giving a 2^{n+2} byte address space. Figure 4 shows an interface using a 8 bit wide device.

Notice that in the diagram there is no clock interface, as is indicated by the term asynchronous. The EMIF still uses the internal clock to coordinate the timing of its signals, however, the Flash responds to the signals at its inputs irrespective of any clock.

Table 4. EMIF Asynchronous Interface Pins

EMIF Signal	Flash Signal	Function
/AOE	/OE	Output Enable - active low during the entire period of a read access.
/AWE	/WE	Write Enable - active low during a write transfer strobe period.
/ARE	N/A	Read Enable - active low during a read transfer strobe period. Although not connected to Flash memory, still used logically to determine when the data is read by the EMIF.
ARDY	RY/BY#	Ready input used to insert wait states into the memory cycle. Hardware method of determining if Flash memory is currently in Program Cycle or Erase Cycle. RY/BY# high indicates device is ready for next operation. Low indicates that device is busy in either Program or Erase cycle. (Not on all devices)
N/A	/BYTE	For 8/16 bit devices, determines if the device will be used in byte mode or in double byte mode. /Byte low selects byte mode. /Byte high selects double byte mode. (Not on all devices)

Figure 3. EMIF-x8/16 bit Flash Interface with ARDY Interface (16 bit mode)

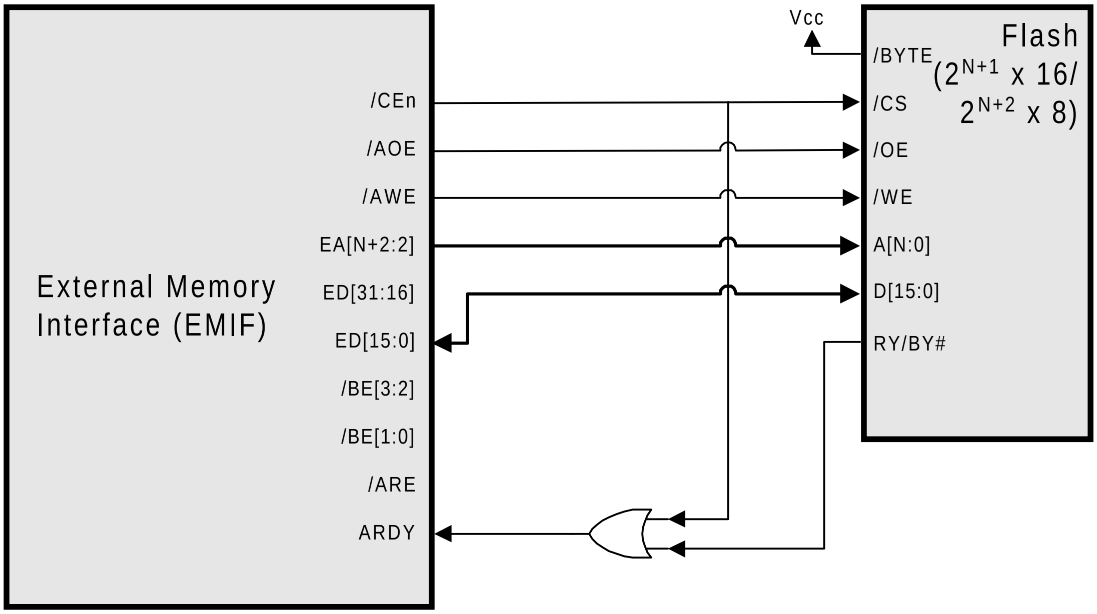
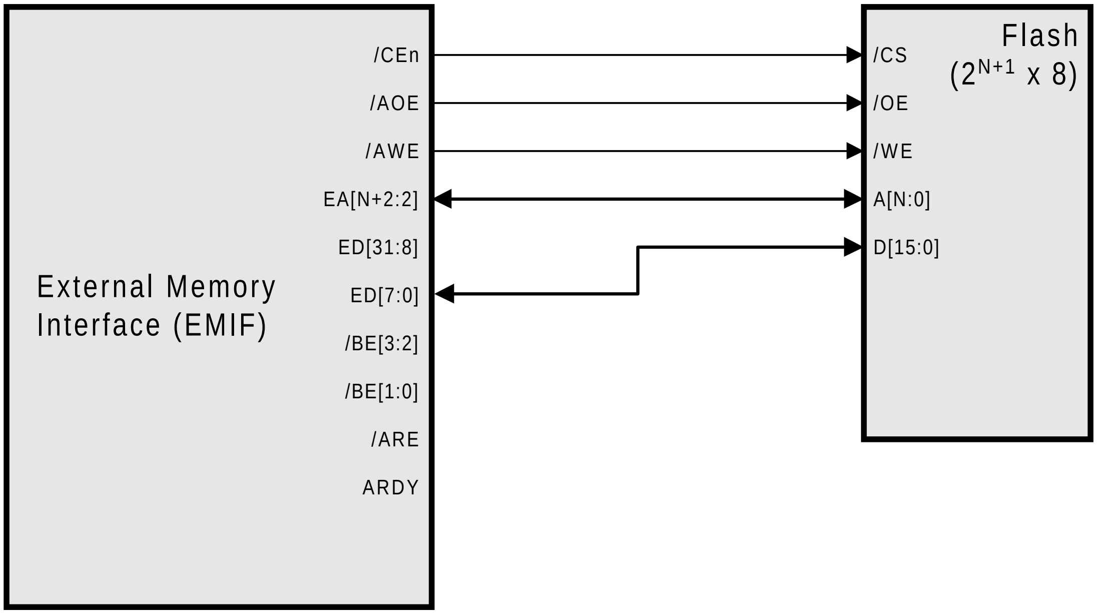


Figure 4. EMIF-8 bit Flash Interface without ARDY Interface



Flash Functionality and Common Commands

Flash is a read-only memory device that has the capability of being reprogrammed in a target system, providing the user with a cost efficient means of maintaining a system that may require code or data changes in the future. Typically, Flash read cycles are compatible with the asynchronous timing which is provided by the C6x. To write data to Flash is more complex, requiring a sequence of writes to command registers in order to execute a command.

Table 5 lists a few of the common JEDEC compatible commands for x8/x16 devices. Devices which are x8 only have a slightly different programming scheme, which is shown in Table 6. The user should verify the programming codes for a particular device in the appropriate data sheet.

Table 5. Flash Commands for TI's TMS29LF800 devices†

Command	Cycles	First		Second		Third		Fourth		Fifth		Sixth	
		Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read	1	RA	RD										
Reset	1	X	F0h										
Chip Erase (8 bit)	6	2aa	aa	555	55	2aa	80	2aa	aa	555	55	2aa	10
Chip Erase (16 bit)	6	555	aa	2aa	55	555	80	555	aa	2aa	55	555	10
Program (8 bit)	4	2aa	aa	555	55	2aa	a0	PA	PD				
Program (16 bit)	4	555	aa	2aa	55	555	a0	PA	PD				

† This programming format was also common to many of the x8/x16 devices, but the programming codes for a particular device should be verified before using.

RA = Read Address

RD = Read Data

PA = Program Address

PD = Program Data

Table 6. Flash Commands for TI's TMS29LF040 devices†

Command	Cycles	First		Second		Third		Fourth		Fifth		Sixth	
		Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read	1	RA	RD										
Reset	1	X	F0h										
Chip Erase (8 bit)	6	5555	aa	2aaa	55	5555	80	5555	aa	2aaa	55	5555	10
Program (8 bit)	4	5555	aa	2aaa	55	5555	a0	PA	PD				

† This programming format was also common to many of the x8 devices, but the programming codes for a particular device should be verified before using.

RA = Read Address

RD = Read Data

PA = Program Address

PD = Program Data

Read/Reset Command

The read mode is activated by issuing the reset command to the Flash memory. The device remains in this mode until another valid command sequence is input in the command register. The device is in this mode by default upon hardware reset, or enters this mode if an invalid command sequence is entered. Also, when any chip or erase operation finishes, the device returns to read mode.

Chip Erase Command

Chip erase is a six bus cycle command sequence, which must be executed if reprogramming a device. This command physically erases the entire address space. Before data can be programmed to the device, the addresses to be programmed must be erased, either by erasing the whole chip or the sector in which the addresses reside. After the last rising edge of /WE, the chip erase operation begins, and any further commands written to the device are ignored until the erase operation is complete.

Program Command

The program command is writes new data to the device. For each byte, double byte, or word to be programmed, first a three write cycle sequence is issued, followed by the address and data to actually be written to the device. The rising edge of /WE starts the program operation. Any further commands written to the device during the program operation are ignored until the program operation is complete

Other Commands

Although the commands listed above are common to most devices, some devices may have their own version of these commands which may or may not be JEDEC compatible. Other common commands include a *sector erase command* which allows the user to erase only a single sector of the Flash memory. This command would take less time than erasing the whole device and is useful if only a small amount of data or code is going to be changed. Another common feature is known as *unlock bypass*, which allows 2 cycle write commands, rather than the 4 cycle writes which are standard. Other less common features include *erase suspend/erase resume*, which allow the erase command to be paused in a given sector, so that reads or writes to another sector can be performed. With this command, the sector being erased cannot be read or written to until the erase operation is complete.

Device Status

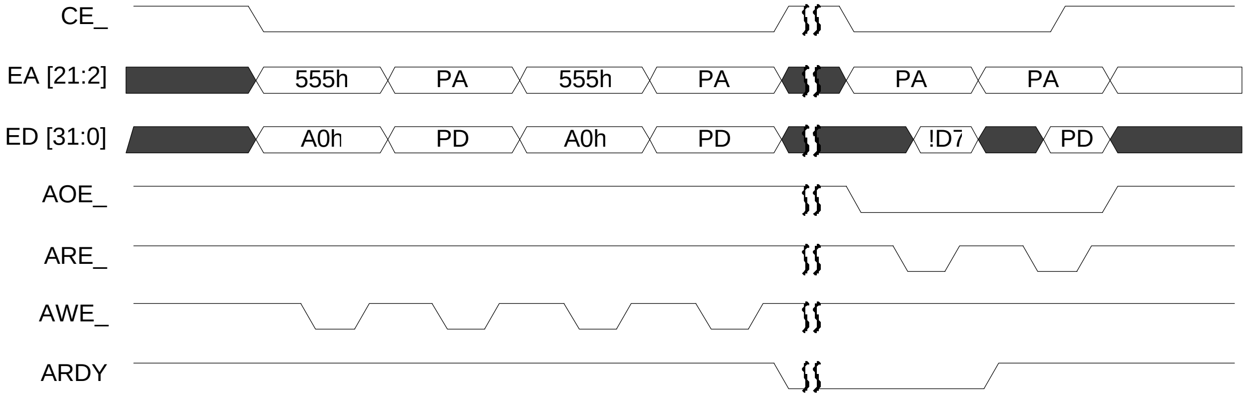
The status of the device during a program operation or chip erase operation can be determined by software, or by hardware and software, depending on the functionality of the desired Flash memory.

Software monitoring can be done by reading the state of the D[7] pin, which is called the *data polling pin*. The status of this bit during a program operation is the complement of the bit written to this pin during the program command, until the program operation completes. When the program operation completes a data read would return the correct data on all pins, including D[7]. During an erase operation, the data read on this pin will be a zero. When the erase operation is complete, a read would return a one on D[7]. Depending on the functionality of the Flash, additional pins would be used in combination to define the function currently in operation.

Figure 5 shows a waveform illustrating a program command sequence followed by data polling to determine when the program operation completes. This diagram assumes that either the RY/BY# signal is NOT connected to the ARDY pin of the C6x, or the flash device does NOT support RY/BY# functionality. The RY/BY# signal is shown to illustrate the beginning and end of the program operation. As seen, before the program operation is completed, the RY/BY# signal is low and a read from the address just programmed returns the complement of the programmed data at bit 7. When the program operation completes, a read will return valid data, indicating that the device is ready for the next operation.

If the flash cannot finish a program or erase operation for some reason, the D[5] pin can be used to detect a time-out condition. A value of 1 on D[5], while D[7] is still not equal to data, indicates a time out. To execute any more commands, a reset command must be issued to the Flash.

Figure 5. Program Command without ARDY Interface - Software Monitoring



Program Command using Data Polling

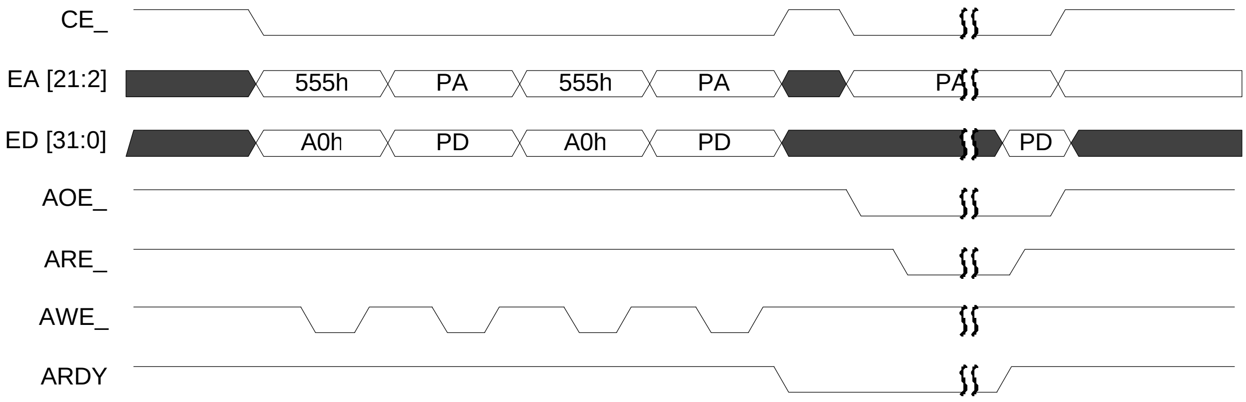
Most Flash memory devices have this data polling functionality, whether or not they include the RY/BY# pin. However, when the RY/BY# pin is connected to the ARDY pin of the ‘C6x, data polling will not be possible, since if the Flash device is busy, indicated by a low level on the RY/BY# pin and the ARDY pin of the C6x, the ‘C6x will not be able to complete a read to the Flash address space until the flash memory completes its current operation and RY/BY# is sent high.

If the device being used has the RY/BY# function, then no software intervention is necessary. A low level on this pin informs the EMIF to extend any future read or write cycles as long as necessary until the state of the pin changes, informing the EMIF that the flash is ready.

Figure 5 shows a waveform illustrating a program command sequence, followed by a read to the same address location. This figure assumes that the RY/BY# signal is connected to the ARDY input of the C6x. The read cycle is extended as long as there is a low level on the ARDY pin. Although this example uses a read cycle to illustrate the ARDY operation, if the next cycle after the program command were a write, the same cycle extension would occur. Therefore, the C6x can begin writing the next program sequence at any time without any software control, since the EMIF will prevent any additional reads or writes until the ARDY signal goes high.

Although the RY/BY# signal allows for a simple interface to the EMIF, there is a limitation in the event that the operation is unable to complete. If software polling were being used without the ARDY interface, this condition could be detected by monitoring the D[5] pin. With the ARDY interface this condition cannot be detected since if the operation cannot complete, the ARDY input will be low, and a read will not be able to be complete to the Flash. This could be worked around by forcing a hardware reset if the ARDY signal is low for a specified number of cycles.

Figure 6. Program Command w/ ARDY Interface - Hardware Monitoring



Program Command using RY/BY#

Programmable ASRAM Parameters

The EMIF allows a high degree of programmability for shaping asynchronous accesses. The programmable parameters that allow this are:

- ☐ **Read Setup / Write Setup :** The time between the beginning of a memory cycle (/CE low, address valid) and the activation of the read or write strobe
- ☐ **Read Strobe / Write Strobe :** The time between the activation and deactivation of the read (/ARE) or write strobe (/AWE)
- ☐ **Read Hold / Write Hold :** The time between the deactivation of the read or write strobe and the end of the cycle (which may be either an address change or the deactivation of the /AOE signal)

These parameters are programmable in terms of CPU clock cycles via fields in the EMIF CE Space Control Registers. Separate setup, strobe, and hold parameters are available for read and write accesses. The SETUP, HOLD, and STROBE fields represent actual cycle counts, in contrast to SDRAM parameters which are the cycle counts - 1.

For either single accesses by the CPU or the first access in a DMA burst to asynchronous memory, the SETUP time observed will be a minimum of 2 cycles. For future accesses in a DMA burst, the SETUP time will be a minimum of 1 cycle.

The STROBE fields have minimum count of 1, and because of this a value of 0 in these fields will still be interpreted as a 1 by the C6x.

HOLD, on the other hand, may be set to 0 cycles. On single accesses by the CPU or the last access in a DMA burst, /CE stays low between 3 and 7 cycles, depending on the STROBE and HOLD parameters.

Figure 7 and Figure 8 help explain the SETUP, HOLD, and STROBE parameters and Table 7 and Table 8 define the other constraints used in these figures.

Table 7. EMIF Input Timing Requirement Definitions

Timing Parameter	Definition
t_{su}	Data Setup time, read D before CLKOUT1 high

Table 8. ASRAM Input Timing Requirement Definitions

Timing Parameter	Definition
t_{acc}	Access Time, from EA, /BE, /AOE, /CE active to ED Valid
t_{xw}	Setup Time from Control/Data Signals Active to /AWE inactive
t_{wp}	Write Pulse Width
t_{wph}	Write Pulse High width
t_h, t_{wr}	Maximum of either Write Recovery Time or Data Hold Time
t_{rc}	Length of the read cycle

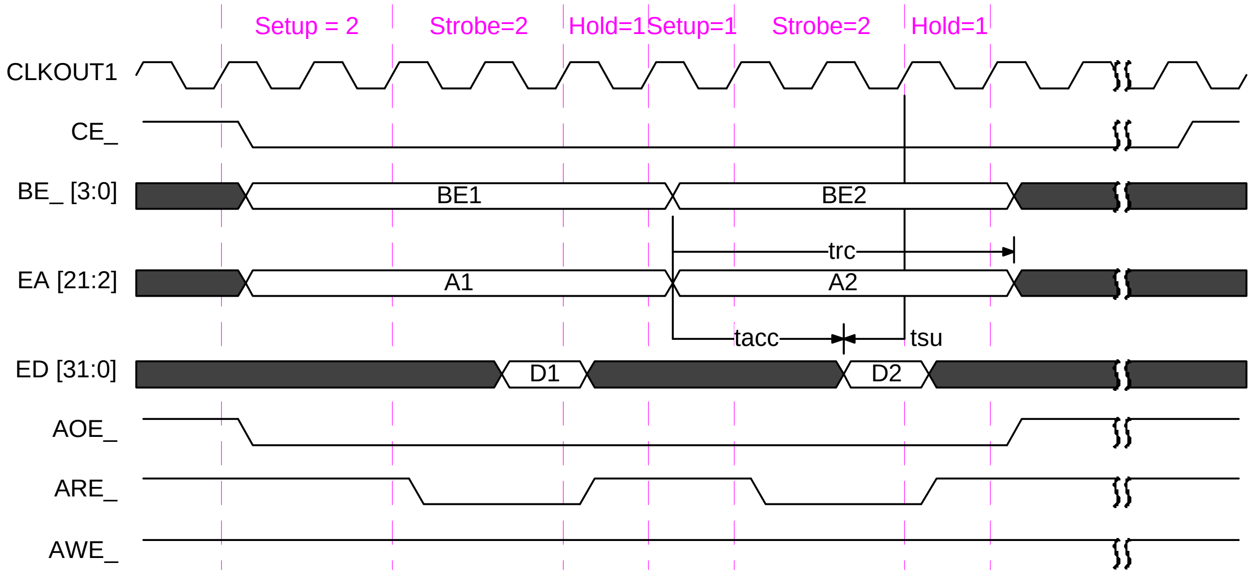
t_{wc}	Length of the write cycle

Asynchronous Reads

Figure 7 illustrates an asynchronous read cycle. This example uses a setup/strobe/hold timing of 1/2/1. An asynchronous read proceeds as follows:

- At the beginning of the setup period
    - /CE becomes active low.
    - /AOE becomes active low.
    - EA becomes valid.
- At the beginning of a strobe period
    - /ARE is pulled active low.
- At the beginning of a hold period
    - /ARE is pulled inactive high
    - Data is sampled on the CLKOUT1 rising edge concurrent with the beginning of the hold period (end of the strobe period), just prior to the /ARE low-to-high transition.
- At the end of the hold period.
    - /AOE becomes inactive as long as another read access to the same /CE space is not scheduled for the next cycle.
    - If an access to the same CE space is not scheduled for the next cycle, /CE will go inactive after 3 to 7 cycles, depending on the parameter settings.

Figure 7. Asynchronous Read Timing Example



ASYNCHRONOUS MEMORY READ TIMING (1/2/1 Timing)

Setting the Read Parameters for a Specific Flash Memory

Notice in Figure 7 that the actual timing that the C6x uses to determine when read data is valid is based on the /ARE signal. In other words, data is actually read on the rising clock edge corresponding to the cycle prior to which /ARE goes high, which is the end of the STROBE period. However, Table 4 shows that /ARE is not connected to Flash memory. This is being pointed out to stress the significance of the SETUP, STROBE, and HOLD times for the C6x and compare them to the significant timing parameters of actual Flash memory.

Flash memory is not synchronized to any clock, however it does have a maximum access time (t_{acc}) which relates when the output data is valid after receiving the required inputs. Thus, the data should be sampled at a time t_{acc} plus t_{su} after the inputs are valid, which, as mentioned, should correspond to the end of the strobe period.

Therefore, when defining the parameters for the C6x for SETUP, STROBE, and HOLD, the following constraints apply:

(1) SETUP + STROBE $\geq t_{acc} + t_{su}$

(2) SETUP + STROBE + HOLD $\geq t_{rc}$

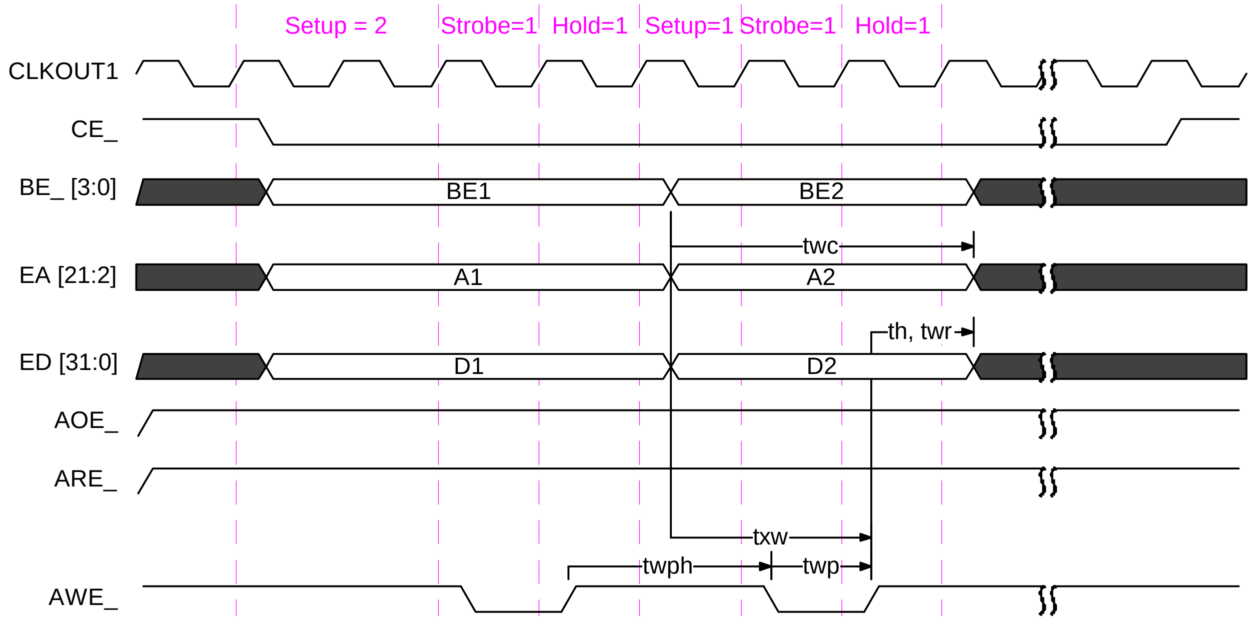
Normally, SETUP can be set to 1 cycle, then STROBE can be solved for using constraint (1). Then, HOLD can be solved for using constraint (2). Of course, the smallest value possible should be used for all three parameters to satisfy the constraints, since normally speed is an important consideration when accessing memory.

Asynchronous Writes

Figure 8 illustrates back to back asynchronous write with a setup/strobe/hold of 1/1/1. An asynchronous write proceeds as follows.

- At the beginning of the setup period
  - /CE becomes active low.
  - /BE[3:0] becomes valid.
  - EA becomes valid.
  - ED becomes valid.
- At the beginning of a strobe period
  - /AWE becomes active low.
- At the beginning of a hold period
  - /AWE becomes inactive high
  - Data is sampled on the CLKOUT1 rising edge concurrent with the beginning of the hold period (end of the strobe period), just prior to the /AWE low-to-high transition.
- At the end of the hold period.
  - ED becomes tri-state only if another write access to the same /CE space is NOT scheduled for the next cycle.
  - If an access to the same CE space is not scheduled for the next cycle, /CE will go inactive after 3 to 7 cycles, depending on the parameter settings.

Figure 8. Asynchronous Write Timing Example



ASYNCHRONOUS MEMORY WRITE TIMING (1/1/1 Timing)

Setting the Write Parameters for a Specific Asynchronous SRAM

For a ASRAM write, the SETUP, STROBE, and HOLD parameters must be set according to the following constraints:

(3) $\text{STROBE} \geq t_{wp}$

(4) $\text{SETUP} + \text{STROBE} \geq t_{xw}$

(5) $\text{HOLD} \geq t_h, t_{wr}$

(6) $\text{HOLD} + \text{SETUP} \geq t_{wph}$

(7) $\text{SETUP} + \text{STROBE} + \text{HOLD} \geq t_{wc}$

Extended Setup Time Asynchronous Reads and Writes

Notice in both Figure 7 and Figure 8, that for the first cycle of a DMA access (or for any signle access) the setup time has a minimum value of 2. In these examples, although a setting of 1 was used for SETUP, the first cycle will still display a setup time of 2 CLKOUT1 cycles.

MTYPE Setting

The MTYPE setting is capable of specifying different interfaces to asynchronous memory: 8 bit ROM, 16 bit ROM, or 32 bit Asynchronous. Depending on how the flash is being used will dictate which of these modes should be selected.

During normal read operation, the MTYPE field should be set to either 8 bit ROM or 16 bit ROM, depending on the specific Flash being used. When either ROM mode is selected, the EMIF will automatically shift the address appropriately so that the correct data will be accessed.

However, during FLASH programming, it is simpler to use the 32 bit interface, and treat the FLASH as if it were 32 bits wide.

Full Example for programming TI's TMS29LF800-90 (ARDY interface)

This section will walk through the configuration steps required to implement TI's TMS29LF800-90, which is an 8 Mbit part organized as 1M x 8 bit or 512k x 16 bit, depending on the state of the mode select pin.

For this implementation the following assumptions will be made :

- ❏ Flash will be used in address space CE1, configured as 16 bit wide ROM.
- ❏ Clock Speed of 200 MHz, therefore t_{period} is 5 ns.
- ❏ The interface will utilize the RY/BY# function of the Flash memory, therefore no software polling will be needed.

Hardware Interface

The hardware interface with the TMS29LF800 Flash memory will be identical to Figure 3, with the CE1 signal used. As seen in the figure, the CE1 output is logically OR-ed with the RY/BY# signal from the flash memory before being tied to the ARDY input of the EMIF. This is done to prevent this signal from interfering with access to any other asynchronous memory in other CE spaces. For example, if a program sequence is written to the flash, the only way the EMIF can recognize that RY/BY# signal is low, is if an access to CE1 is done, allowing a low signal to pass on to ARDY, and extending the cycle until the Flash is finished with the program operation. If after the program sequence, an access to CE2 is done instead, then since CE1 is still high, the input to ARDY will be high, and not inhibit access to CE2.

The reason CE1 is selected for the memory space in this example is due to the boot processes supported by the 'C6x. The 'C6x can be set up to transfer data from CE1 to address 0 with the DMA immediately after reset, which is a very good use for flash memory. If flash is used at CE1, then semi-permanent boot code can be stored there. The state of the Bootmode pins at reset let the processor know what type of memory is located at CE1.

Register Configuration

Table 9 and Table 10 summarize the timing characteristics of the TMS29LF800-90, which will be used to calculate the values for the CE0 Space Configuration Register. This data was taken from the C6x Data Sheet and the TMS29LF040 Data Sheet.

Table 9. EMIF Input Requirements from ASRAM for TMS29LF800-90

NO			MIN	MAX	UNIT
1	t_{su}	Setup time, read D before CLKOUT1 high0	5		ns

Table 10. FLASH Input Requirements from EMIF for TMS29LF800-90

NO			MIN	MAX	UNIT
2	t_{acc}	Access Time, from EA, /BE, /AOE, /CE active to ED Valid		90	ns
3	t_{xw}	Time from Control/Data Signals Active to /AWE inactive	50		ns
4	t_{wp}	Write Pulse Width	50		ns
5	t_{wph}	Write Pulse High width	30		ns
6	t_h, t_{wr}	Maximum of either Write Recovery Time or Data Hold Time	0		ns
7	t_{rc}	Length of the read cycle	90		ns
8	t_{wc}	Length of the write cycle	90		ns

Read Calculations

- **SETUP = 1**, based on assumptions stated in the section, Setting the Read Parameters for a Specific .

- SETUP + STROBE $\geq t_{acc} + t_{su}$

  Therefore,

  STROBE $\geq (t_{acc} + t_{su})/\text{tperiod} - \text{SETUP}$

  $\geq$ (90 ns + 5 ns)/ 5 ns -1

  $\geq$ 19 cycles - 1 cycle = 18 cycles

  STROBE = 18 cycles

- SETUP + HOLD + STROBE $\geq t_{rc}$

  Therefore,

  HOLD $\geq t_{rc}$ – SETUP - STROBE

  $\geq$ 90ns/5ns - 1 -18 = -1

  HOLD = 0 cycles, since it cannot be negative.

Write Calculations

☐ STROBE $\geq t_{wp}$

$\geq t_{wp} / t_{period}$ = 50 ns / 5ns = 10 cycles

STROBE = 10 cycles

☐ SETUP + STROBE $\geq t_{xw}$

Therefore,

SETUP $\geq t_{xw}$ /tperiod - STROBE = 50ns / 5ns - 2 cycles = 8 cycles.

SETUP = 8 cycle

☐ HOLD $\geq t_h$, t_{wr} = 0

HOLD = 0 cycles

☐ HOLD + SETUP $\geq t_{wph}$

This requirement is satisfied. The sum of the two parameters is 8 cycles, which is greater than 30 ns (6 cycles).

☐ SETUP + HOLD + STROBE $\geq t_{wc}$

This requirement is satisfied. The sum of the three parameters is 18 cycles, which is equal to 90 ns (18 cycles).

MType Setting

The MTYPE field will be set for 32 bit asynchronous interface, since this example is illustrating Flash programming. This will allow us to treat the Flash as a 32 bit wide device, and allow us to increment the program destination pointer by 4 bytes, so that no address shifting will need to be done to program the proper address. The control addresses will still need to be shifted so that the proper address shows up on the address lines. This is illustrated in the sample code in Appendix A.

Using the above calculations, the CE Space Control Register can now be properly configured. Figure 9 shows the CE1 Space Control Register with the properly assigned values for each field. The value to be used is 0x82811220.

Figure 9. EMIF CE1 Space Control Register Diagram for TMS29LF800-90

31 28	27 22	21 20	19 16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
1000	001010	00	0001

15 14	13 8	7	6 4	3 2	1 0
rsv	READ STROBE	rsv	MTYPE	reserved	READ HOLD
rsv	010010	rsv	010	reserved	00

Software Control

Since this interface utilizes the RY/BY# signal as an input to the EMIF via the ARDY input, software polling will not be necessary, since any read or write to the flash will automatically be extended until the Flash is ready to respond. Since software polling is not necessary, the software algorithm will be simple.

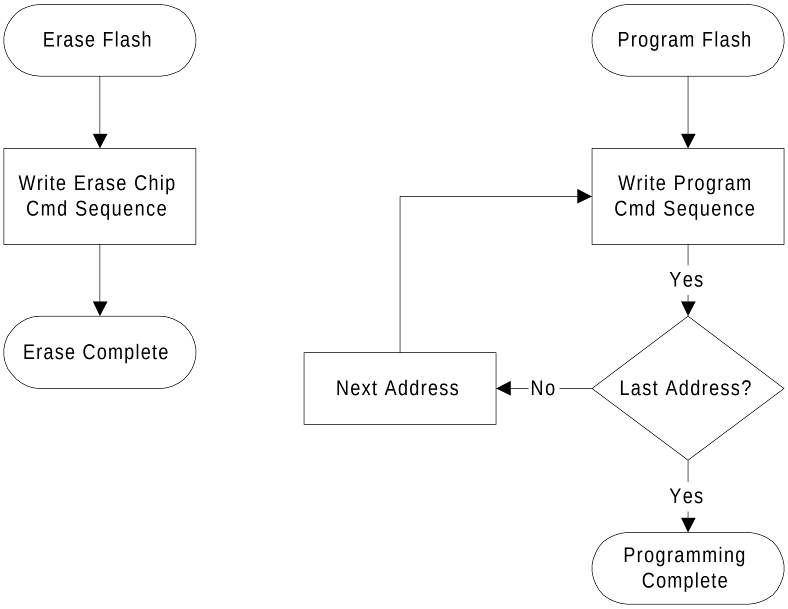
Read Operation

For this flash device, upon hardware reset, the device automatically is in read mode, so no special steps need to be taken if the device is to be used simply for reading code or data. At the first convenient time, the EMIF registers should be set as described above in the Register Configuration section.

Write Operation

The write operation is slightly more involved, but is still simple since no software polling must be done. Figure 10 contains flowcharts of the two operations which are required to write new data to the flash memory. This flowchart and code example will assume that the source data has been written to CE2 via the HPI or some other means, which will not be discussed here. This example will illustrate the erasing, then programming of the Flash memory. For the source code used, refer to Appendix A. Sample code for Programming TMSLF800

Figure 10. Erase Chip and Program Command Flow Charts



Full Example for programming TI's TMS29LF040-15

This section will walk through the configuration steps required to implement TI's TMS29LF040-15, which is a 4 Mbit part organized as 512k x 8 bit.

For this implementation the following assumptions will be made :

- Flash will be used in address space CE1, configured as 16 bit wide ROM.
- Clock Speed of 200 MHz, therefore t_{period} is 5 ns.
- The interface will **not** utilize the RY/BY# function of the Flash memory, therefore software polling will be needed.

Hardware Interface

The hardware interface with the TMS29LF040 Flash memory will be identical to Figure 4, with the CE1 signal used.

The reason CE1 is selected for the memory space in this example is due to the boot processes supported by the 'C6x. The 'C6x can be set up to transfer data from CE1 to address 0 with the DMA immediately after reset, which is a very good use for flash memory. If flash is used at CE1, then semi-permanent boot code can be stored there. The state of the Bootmode pins at reset let the processor know what type of memory is located at CE1.

Register Configuration

Table 11 and Table 12 summarize the timing characteristics of the TMS29LF040-15, which will be used to calculate the values for the CE0 Space Configuration Register. This data was taken from the C6x Data Sheet and the TMS29LF040 Data Sheet.

Table 11. EMIF Input Requirements from ASRAM for TMS29LF040-15

NO			MIN	MAX	UNIT
1	t_{su}	Setup time, read D before CLKOUT1 high0	5		ns

Table 12. FLASH Input Requirements from EMIF for TMS29LF040-15

NO			MIN	MAX	UNIT
2	t_{acc}	Access Time, from EA, /BE, /AOE, /CE active to ED Valid		150	ns
3	t_{xw}	Time from Control/Data Signals Active to /AWE inactive	50		ns
4	t_{wp}	Write Pulse Width	50		ns
5	t_{wph}	Write Pulse High width	20		
6	t_h, t_{wr}	Maximum of either Write Recovery Time or Data Hold Time	0		ns
7	t_{rc}	Length of the read cycle	150		ns
8	t_{wc}	Length of the write cycle	150		ns

Read Calculations

- ☐ **SETUP = 1**, based on assumptions stated in the section, Setting the Read Parameters for a Specific .
- ☐ SETUP + STROBE ≥ $t_{acc} + t_{su}$

  Therefore,

  STROBE ≥ $(t_{acc} + t_{su})$/tperiod - SETUP

  ≥ (150 ns + 5 ns)/ 5 ns -1

  ≥ 31 cycles - 1 cycle = 30 cycles

  STROBE = 30 cycles

- ☐ SETUP + HOLD + STROBE ≥ t_{rc}

  Therefore,

  HOLD ≥ t_{rc} – SETUP - STROBE

  ≥ 150ns/5ns - 1 -30 = -1

  HOLD = 0 cycles, since it cannot be negative.

Write Calculations

❑ STROBE $\geq t_{wp}$

$\geq t_{wp} / t_{period}$ = 50ns / 5ns = 10 cycles

STROBE = 10 cycles ** This value is not used as explained next **

❑ SETUP + STROBE $\geq t_{xw}$

Therefore,

SETUP $\geq t_{xw}$ /tperiod **-** STROBE = 50ns / 5ns **-** 2 cycles = 8 cycles.

SETUP = 8 cycles

❑ HOLD $\geq t_h$, t_{wr} = 0

HOLD = 0 cycles

❑ HOLD + SETUP $\geq t_{wph}$

This requirement is satisfied. The sum of the two parameters is 8 cycles, which is greater than 20 ns (4 cycles).

❑ SETUP + HOLD + STROBE $\geq t_{wc}$

This requirement is NOT satisfied. The sum of SETUP, HOLD, and STROBE is 13 cycles, using the calculated values. However, since twc is 150 ns, we need a total cycle count of 30 cycles. We can extend the write strobe time from 10 to 22 cycles in order to satisfy this constraint.

STROBE = 22 cycles

MType Setting

The MTYPE field will be set for 32 bit asynchronous interface, since this example is illustrating Flash programming. This will allow us to treat the Flash as a 32 bit wide device, and allow us to increment the program destination pointer by 4 bytes, so that no address shifting will need to be done to program the proper address. The control addresses will still need to be shifted so that the proper address shows up on the address lines. This is illustrated in the sample code in Appendix B.

Using the above calculations, the CE Space Control Register can now be properly configured. Figure 11 shows the CE1 Space Control Register with the properly assigned values for each field. The value to be used is 0x8581de20.

Figure 11. EMIF CE1 Space Control Register Diagram for TMS29LF040-15

31 28	27 22	21 20	19 16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
1000	010110	00	0001
RW, +1111	RW, +111111	RW, +11	RW, +1111

15 14	13 8	7	6 4	3 2	1 0
TA	READ STROBE	rsv	MTYPE	reserved	READ HOLD
11	011110	rsv	010	reserved	00
RW, +11	RW, +111111	R, +0	RW, +010	R, +0	RW, +11

Software Control

Since this interface does not utilize the RY/BY# signal as an input to the EMIF via the ARDY input, software polling will be necessary, since the EMIF will not automatically wait until the embedded algorithm is complete.

Read Operation

For this flash device, upon hardware reset, the device automatically is in read mode, so no special steps need to be taken if the device is to be used simply for reading code or data. At the first convenient time, the EMIF registers should be set as described above in the Register Configuration section.

Write Operation

The write operation is slightly more involved. In order to detect if the embedded program or erase algorithm is complete, software polling must be done. Figure 12 contains flowcharts of the two operations which are required to write new data to the flash memory--erase and program. In each of these algorithms, device polling must be done via software. The polling algorithm is shown in Figure 13. This flowchart and code example will assume that the source data has been written to CE2 via the HPI or some other means, which will not be discussed here. This example will illustrate the erasing, then programming of the Flash memory. For source code, see Appendix B.

Figure 12. Erase Chip and Program Command Flow Charts

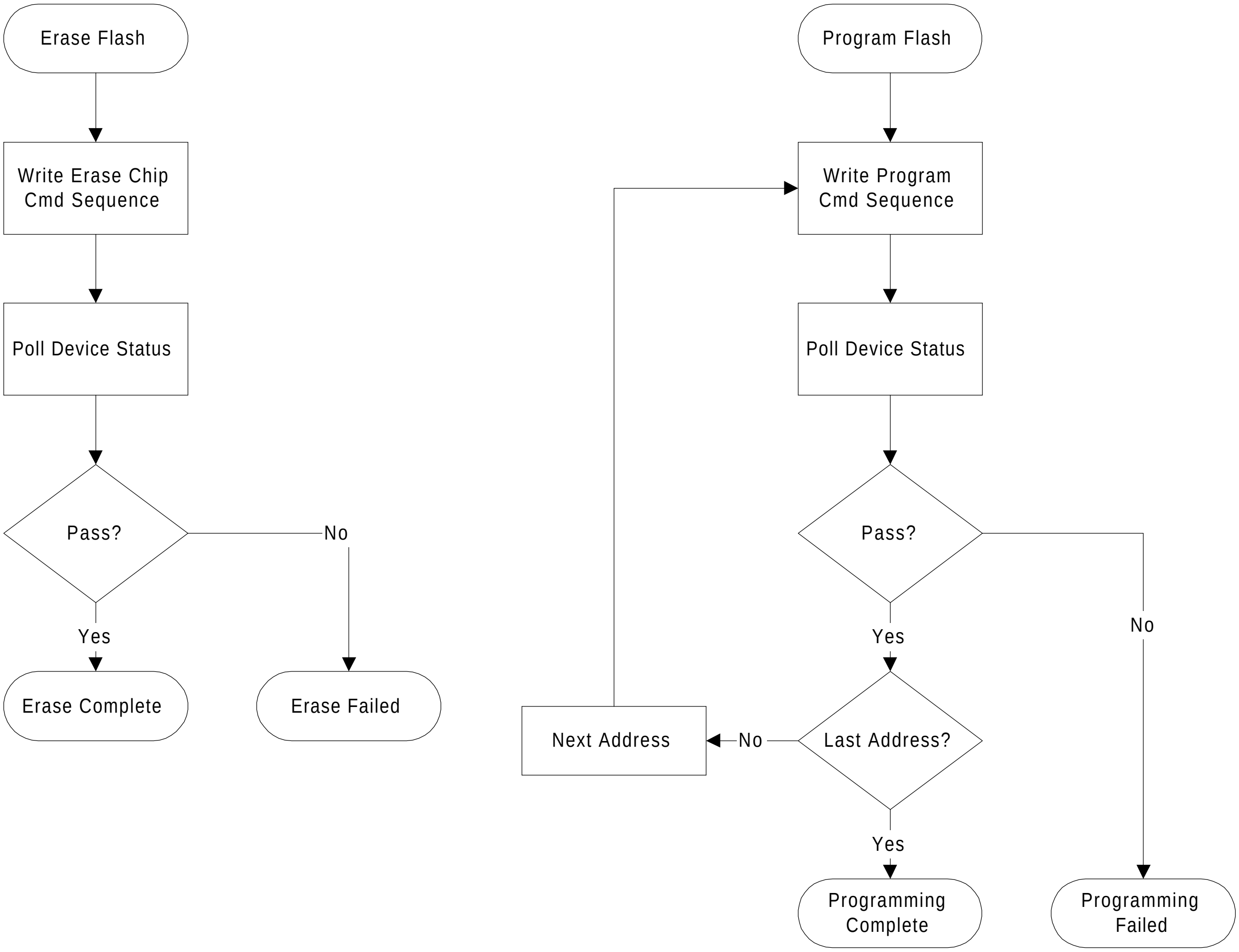
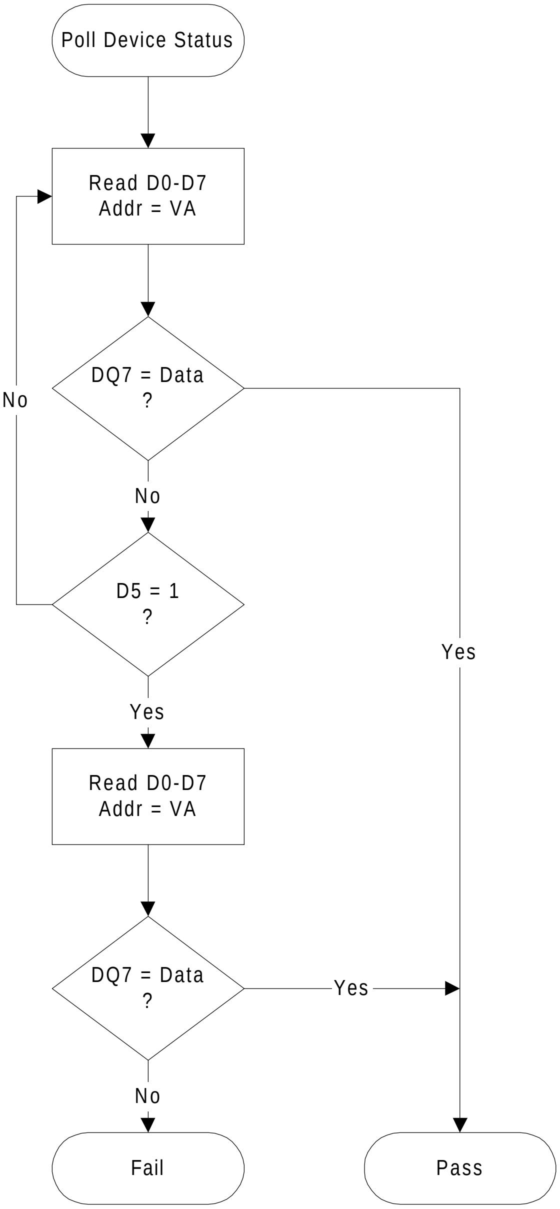


Figure 13. Data Poll Flow Chart



Appendix A. Sample code for Programming TMSLF800 (ARDY Interface)

```
/*******************************************************************************/
/* Hardware.c                                                                  */
/* Kyle Castille                                                               */
/* This program will create a dummy data buffer with an incrementing count    */
/* in internal memory, and then program this data to the TMS28LF040-15 Flash  */
/* which is a 512k x 8, 150 ns Flash memory.                                   */
/*                                                                             */
/* This program assumes that the ARDY interface is used, so no software       */
/* checking is done to detect end of operation for erase or program           */
/*                                                                             */
/*******************************************************************************/
#define     CE1_ADDRS         0x01400000
#define     INT_MEM           0x80000000
#define     CE1_CNTRL         0x01800004
#define     FLASH_ADDRS       CE1_ADDRS
#define     SRC_ADDRS         INT_MEM
#define     LENGTH            0x400
#define     TRUE              1
#define     FALSE             0

#include <stdio.h>
#include <stdlib.h>

void load_source (short * source, int num_words);
void erase_flash(int * flash_addrs);
void program_flash(short * source, int * flash_addrs, int num_words);
void emif_config();

void
main(){
      int * flash_ptr = (int *)FLASH_ADDRS;
      short * src_ptr = (short *)SRC_ADDRS;


      emif_config();
      load_source(src_ptr, LENGTH);
      erase_flash(flash_ptr);
      program_flash(src_ptr, flash_ptr, LENGTH);
      printf("Successful erase and program!!!");

}

/*******************************************************************************/
/* emif_config :Routine to configure the Emif for operation with              */
/*          TMS29LF800-90 at CE1.  This routine sets the CE1 control register */
/*          for a 32 bit asynchronous memory interface with the following     */
/*          parameters:                                                        */
/*          Mtype = 010                                                        */
/*          Read Setup/Strobe/Hold = 1/18/0                                    */
/*          Write Setup/Strobe/Hold = 8/10/0                                   */
/*                                                                             */
/*******************************************************************************/

void emif_config(){

      * (int *)CE1_CNTRL = 0x82811220;

}

/*******************************************************************************/
/* load_source :Routine to load the source memory with data.  This routine    */
/*          loads an incrementing count into the source memory for            */
/*          demonstration purposes.                                            */
/* Inputs:                                                                     */
/*    source_ptr :      Address to be used as the source buffer               */
/*    code_ptr   :      Length to be programmed                                */
/*                                                                             */
/*******************************************************************************/

void load_source(short * source_ptr, int length)
{
      int i;
      for (i = 0; i < length; i ++){
            * source_ptr++ = i;
      }
}

/********************************************************************************/
/* erase_flash : Routine to erase entire FLASH memory TMS29LF800 (1M x 8bit/  */
/*          512k x 16bit)                                                      */
/* Inputs:                                                                     */
/*    flash_ptr: Address of the FLASH PEROM                                    */
/*                                                                             */
/********************************************************************************/

void erase_flash(int * flash_ptr)
{
                              /* Control addresses are left shifted so that   */
                              /* they appear correctly on the EMIF's EA[19:2] */
                              /* Short << 1 == Word                           */
      short * ctrl_addr1 = (short *) ((short)flash_ptr + (0x555 << 1));
      short * ctrl_addr2 = (short *) ((short)flash_ptr + (0x2aa << 1));;

      * ctrl_addr1 = 0xaa;    /* Erase sequence writes to addr1 and addr2     */
      * ctrl_addr2 = 0x55;    /* with this data                         */
      * ctrl_addr1 = 0x80;
      * ctrl_addr1 = 0xaa;
      * ctrl_addr2 = 0x55;
      * ctrl_addr1 = 0x10;
}

/********************************************************************************/
/* program_flash: Routine to program FLASH TMS29LF800                          */
/* Inputs:                                                                     */
/*    flash_ptr: Address of the FLASH PEROM                                    */
/*    code_ptr : Address of the array containing the code to program          */
/*                                                                             */
/********************************************************************************/

void program_flash(short * source_ptr, int * flash_ptr, int length)
```

```
{
      int i;
                              /* Control addresses are left shifted so that   */
                              /* they appear correctly on the EMIF's EA[19:2] */
                              /* Short << 1 == Word                           */
      int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 1));
      int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 1));;

      for (i = 0; i < length; i++){

            * ctrl_addr1 = 0xaa;
            * ctrl_addr2 = 0x55;
            * ctrl_addr1 = 0x80;

            * flash_ptr++ = * source_ptr++;
      }
}
```

Appendix B. Sample Code for Programming TMSLF04

```
/******************************************************************************/
/* Software.c                                                                 */
/* Kyle Castille                                                              */
/* This program will create a dummy data buffer with an incrementing count    */
/* in internal memory, and then program this data to the TMS28LF040-15 Flash  */
/* which is a 512k x 8, 150 ns Flash memory.                                  */
/* This program assumes that the ARDY interface is NOT used, therefore        */
/* monitoring is done to detect end of operation                              */
/*                                                                            */
/******************************************************************************/
#define     CE1_ADDRS         0x01400000
#define     INT_MEM           0x80000000
#define     CE1_CNTRL         0x01800004

#define     FLASH_ADDRS       CE1_ADDRS
#define     SRC_ADDRS         INT_MEM
#define     LENGTH            0x400
#define     TRUE              1
#define     FALSE             0

#include <stdio.h>
#include <stdlib.h>

void load_source (char * source, int num_words);
int erase_flash(int* flash_addrs);
int program_flash(char * source, int * flash_addrs, int num_words);
int poll_data(int *,char );
void emif_config();

void
main(){
      int pass = TRUE;
      int * flash_ptr = (int *)FLASH_ADDRS;
      char * src_ptr = (char *)SRC_ADDRS;
      emif_config();
      load_source(src_ptr, LENGTH);
      pass = erase_flash(flash_ptr);
      if (pass){
            pass = program_flash(src_ptr, flash_ptr, LENGTH);
            if (!pass)
                  printf("Failed in program operation");
            else
                  printf("Successful erase and program!!!");
      }
      else
            printf("Failed in erase operation");

}

/*******************************************************************************/
/* emif_config :Routine to configure the Emif for operation with              */
/*          TMS29LF040-15 at CE1.  This routine sets the CE1 control register */
/*          for a 32 bit asynchronous memory interface with the following     */
/*          parameters:                                                        */
/*          Mtype = 010                                                        */
/*          Read Setup/Strobe/Hold = 1/30/0                                    */
/*          Write Setup/Strobe/Hold = 8/22/0                                   */
/*                                                                             */
/*******************************************************************************/
void emif_config()
{
      * (int *)CE1_CNTRL = 0x8581de20;
}

/*******************************************************************************/
/* load_source :Routine to load the source memory with data.  This routine    */
/*          loads an incrementing count into the source memory for            */
/*          demonstration purposes.                                            */
/* Inputs:                                                                     */
/*    source_ptr :      Address to be used as the source buffer               */
/*    code_ptr   :      Length to be programmed                                */
/*                                                                              */
/********************************************************************************/

void load_source(char * source_ptr, int length)
{
      int i;

      for (i = 0; i < length; i ++){
            * source_ptr++ = (char) i;
      }
}

/********************************************************************************/
/* erase_flash : Routine to erase entire FLASH memory TMS29LF040 (512Kx8bit)  */
/* Inputs:                                                                      */
/*    flash_ptr: Address of the FLASH                                           */
/* Return value:                                                                */
/*    Returns TRUE if passed, or FALSE if failed.  Pass or failure is          */
/*    determined during the poll_data routine.                                  */
/*                                                                              */
/********************************************************************************/

int erase_flash(int * flash_ptr)
{
                              /* Control addresses are left shifted so that   */
                              /* they appear correctly on the EMIF's EA[19:2] */
                              /* Byte << 2 == Word                            */
      int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
      int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));


      int pass = TRUE;

      * ctrl_addr1 = 0xaa;    /* Erase sequence writes to addr1 and addr2     */
      * ctrl_addr2 = 0x55;    /* with this data                               */
      * ctrl_addr1 = 0x80;
      * ctrl_addr1 = 0xaa;
      * ctrl_addr2 = 0x55;
      * ctrl_addr1 = 0x10;

      pass = poll_data(flash_ptr, (char) 0xff);
      if (!pass)
            printf("failed erase\n\n");
      return pass;
}

/*******************************************************************************/
/* program_flash: Routine to program FLASH TMS29LF040(512K x 8bit)            */
/* Inputs:                                                                     */
/*    flash_ptr: Address of the FLASH PEROM                                    */
/*    code_ptr : Address of the array containing the code to program          */
/* Return value:                                                               */
/*    Returns TRUE if passed, or FALSE if failed.  Pass or failure is         */
/*    determined during the poll_data routine.                                 */
/*                                                                             */
/*******************************************************************************/

int program_flash(char * source_ptr, int * flash_ptr, int length)
{
      int i;
      char data;
      int pass = TRUE;

      int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
      int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));

      for (i = 0; i < length && pass == TRUE; i++){
            * ctrl_addr1 = 0xaa;
            * ctrl_addr2 = 0x55;
            * ctrl_addr1 = 0x80;

            data = * source_ptr++;
            * flash_ptr++ = (int) data;

            pass = poll_data(flash_ptr, data);
      }
      if (!pass)
            printf("Failed at address %x \n\n", (int) flash_ptr);
      return pass;
}
/******************************************************************************/
/* poll_data: Routine to determine if Flash has successfully completed the    */
/*          program or erase algorithm.  This routine will loop until         */
/*          either the embedded algorithm has successfully completed or       */
/*          until it has failed.                                              */
/*                                                                            */
/* Inputs:                                                                    */
/*    prog_ptr : Address just programmed                                      */
/*    prog_data: Data just programmed to flash                                */
/* Return value:                                                              */
/*    Returns TRUE if passed, or FALSE if failed.                             */
/*                                                                            */
/******************************************************************************/

int poll_data(int * prog_ptr, char prog_data)
{
      char data;
      int fail = FALSE;
      char d7;

      do {
            data = (char) * prog_ptr;
            if (data != prog_data) {                  /* is D7 != Data? */
                  if ((data & 0x20) == 0x20) {        /*is D5 = 1 ?           */
                        data = (char) * prog_ptr;
                        if (data != prog_data)        /* is D7 = Data?  */
                              fail = TRUE;
                        else
                              return TRUE;            /* PASS */
                  }
            }
            else
                  return TRUE;                        /*PASS*/
      } while (!fail);
```

```
    return FALSE;                                    /* FAIL */
}
```

References

“TMS320C6201 Data Sheet, Revision 2,” Texas Instruments, October 1997.

“TMS320C62xx Peripherals Reference Guide,” Texas Instruments, October 1997.

“TMS29LF800 Data Sheet,” Texas Instruments, October 1997.

“TMS29LF040 Data Sheet,” Texas Instruments, October 1997.