PalmMicro
Communications, Inc.

PA-1680

Datasheet (Version 1.1)

© 1999-2000 PalmMicro Commnications, Inc.
630 Alder Dr. Milpitas, CA 95035

Revision History

Revision	1.1.1.1.1Date	Name	Revision contents
1.0	11/12/2000	H.Y. Zhang	First revision
1.1	12/4/2000	H.Y. Zhang	Programming guide
1.2	3/20/2000	H.Y. Zhang	Operating condition

Table of contents

1. Features 4
1.1 Main Applications 4
1.2 Chip Specs 4
1.3 System Level Support 4
1.4 Pin Description 5
2. Chip Architecture 7
2.1 Controller Subsystem 8
2.1.1 Controller Core 8
2.1.2 Program Memory 8
2.1.3 Data Memory 9
2.2 DSP Subsystem 10
2.2.1 DSP Core 11
2.2.2 DSP Memories 12
2.2.3 DSP Memory Mapped Registers 13
2.3 CmdQueue 13
2.4 System Memory Interface 14
2.4.1 External Memory Support 15
2.4.2 DMA Channel Arbitration 15
2.4.3 External Memory Interfaces 16
2.4.4 DMA Setup 16
2.4.5 BYTE access mode 17
2.4.6 FIFO's and interface to SDRAM controller 17
2.5 Data Flash Interface 19
2.6 LCD Interface 19
2.7 Camera Interface 20
2.8 Codec Interface 21
2.8.1 AC97 21
2.8.2 UDA 21
2.8.3 Codec FIFO 22
2.9 USB Interface 22
2.10 Programmable I/O, Keypad, Interrupts and Timer 23
2.10.1 Programmable I/O 24
2.10.2 Keypad 24
2.10.3 Interrupts 24
2.10.4 Timer 25
2.11 Clocks 25
3. Programming Guide 27
3.1 DMA 27
3.1.1 Overview 27
3.1.2 DMA Basics 27
3.1.3 Initialization 28
3.1.4 Setup DMA to/from Controller Buffer 29
3.1.5 Setup DMA to /from DSP (PM and DMX) 29
3.1.6 DMA to/from FIFO 30
3.1.7 BYTE access mode 31
3.1.8 Interrupts and Status 31
3.1.9 DSP Direct Access to DMA 31
3.2 Sensor Interface Programming 32
3.2.1 Control Signals 32

3.2.2 Defining an active region 32
3.2.3 Camera I/O and I2C interface 33
3.2.4 FIFO 33
3.3 Codec Interface Programming 33
3.3.1 Codec Types 33
3.3.2 UDA 33
3.3.3 AC97 34
3.3.4 DMA Channels 34

1. Features

- 8051-instruction-set compatible controller core
- 2181-instruction-set compatible DSP co-processor
- High-speed bulk-mode USB interface
- Parallel port for digital image/video input
- ADC/DAC: AC97, I2S (Philips)
- Flash Memory support:
  - 32MB Data FLASH
  - SmartMedia card
  - LCD interface
- System Memory support:
  - Up to 64MB SDRAM
  - Up to 512KB SRAM
  - External glue logic to Bluetooth RF, etc.
- Program Memory support:
  - Up to 1MB EEPROM, SRAM, or FLASH
- Power-down and power-saving modes
- Up to 17 general purpose I/O pins
- Keypad support (up to 12 keys directly or up to 128 keys with an external encoder)
- Special mode for Handspring interface

1.1 Main Applications

- Standalone Portable Multimedia Device
  - MP3 Player
  - Digital Camera
  - Digital Voice Recorder
  - IP-Phone
- Add-on Module
  - Handspring

1.2 Chip Specs

Technology: 0.25u process 2.5V
Speed: Up to 50MHz
Package: 160/128 PQFP
Recommended Operating Conditions:

Symbol	Description	Min	Max	Units
VCC	Input core voltage relative to GND	2.5 – 5%	2.5 + 5%	V
VCC3 (*)	Input I/O voltage relative to GND	1.6	3.9	V

(*) Notes: This voltage range is only valid when the USB pads are not used. In case of USB applications, the working voltage is 2.2V ~ 3.3V.

1.3 System Level Support

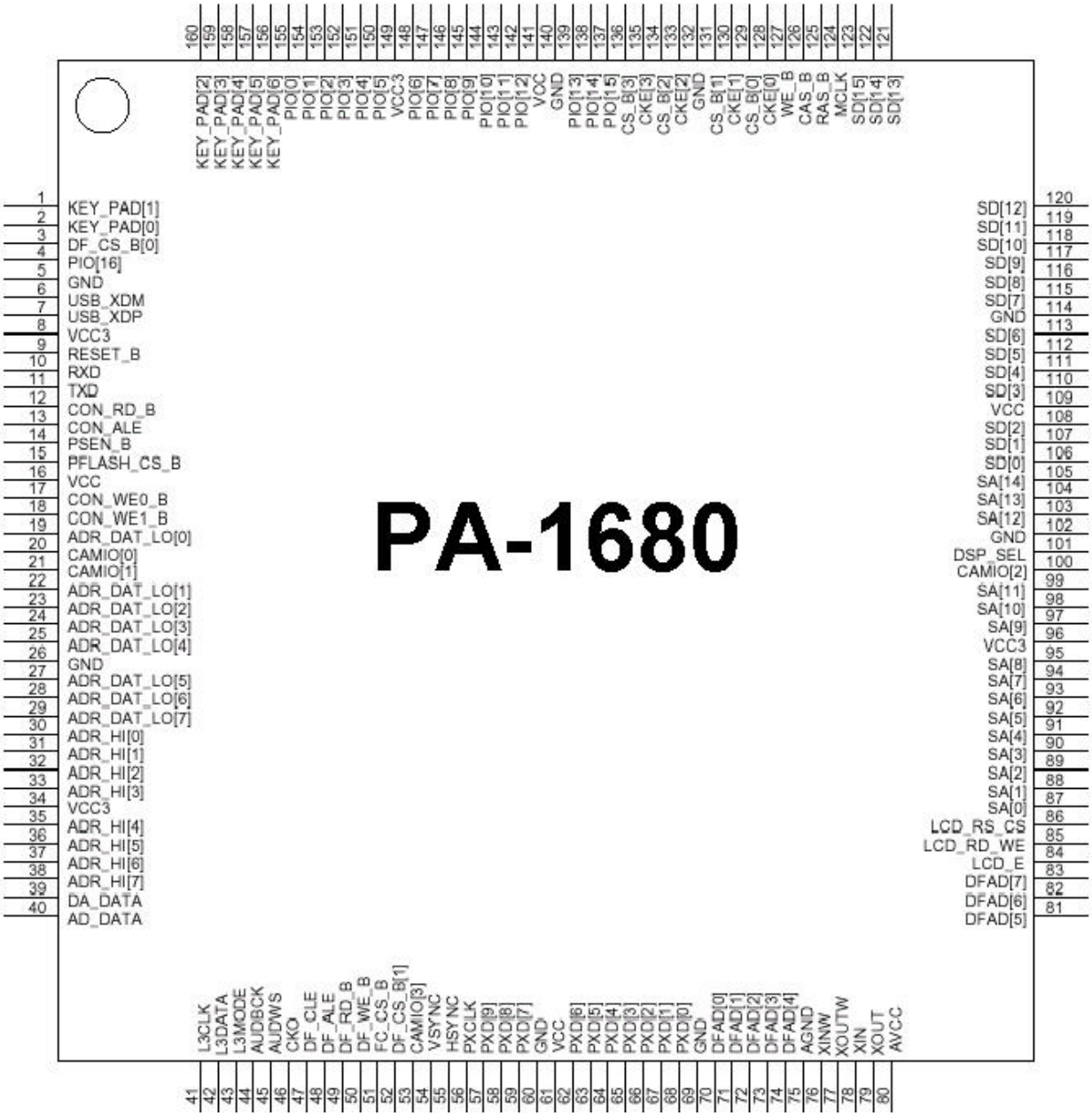
External Memory:

Smart Media Card (up to 128MB)
Up to 32MB of Data FLASH

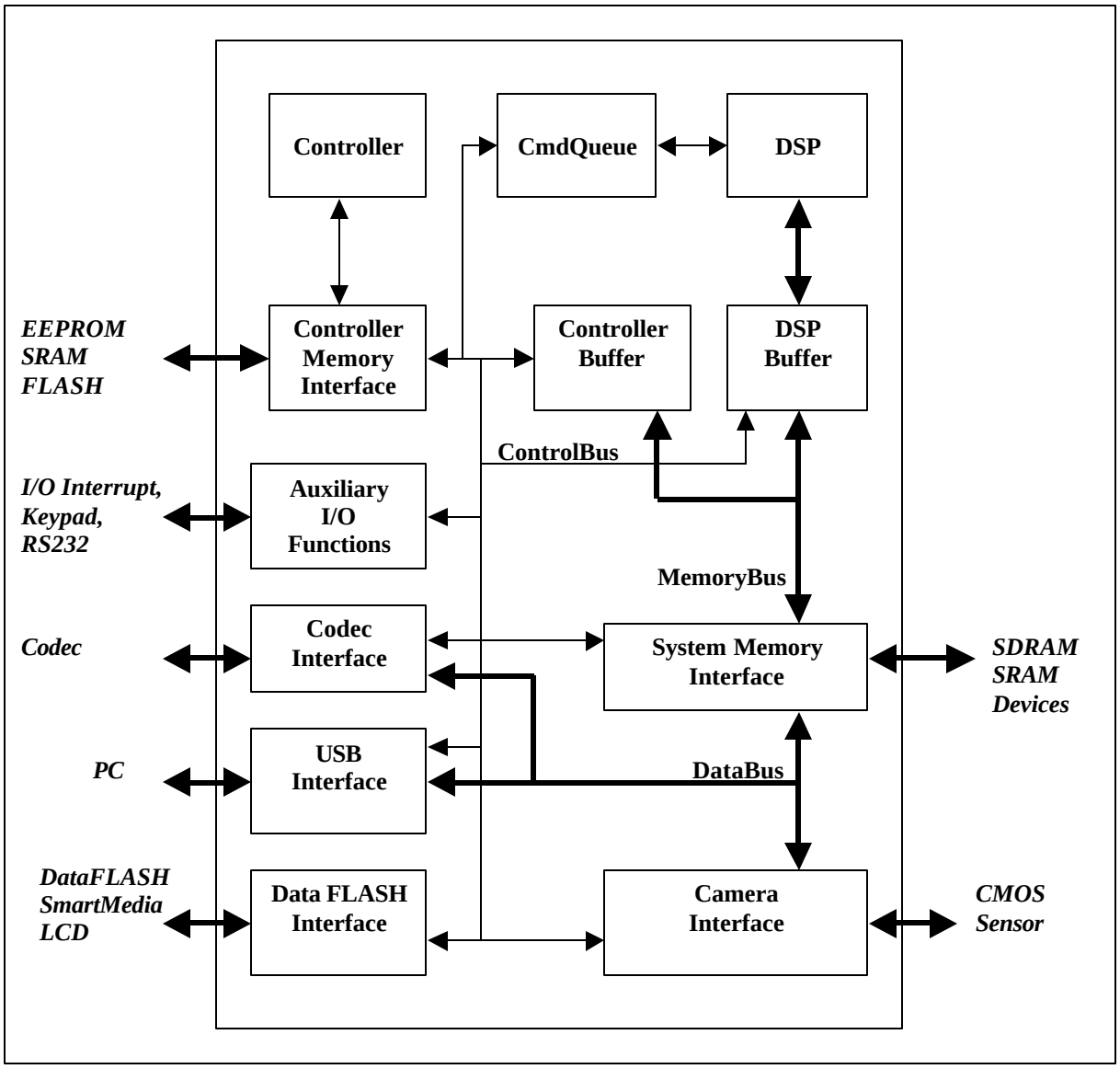
Up to 64MB of SDRAM or 512KB of SRAM
Up to 1MB EEPROM/SRAM/FLASH for program/data

Others: CMOS Sensor Chip as Video-input
Analog input from Microphone
Low-power DAC for digital audio
RF Chip w/ glue logic
Keypad/Keypad encoder
LCD display and controller
Power monitor and regulators

1.4 Pin Description


PA-1680
1 KEY_PAD[1]
2 KEY_PAD[0]
3 DF_CS_B[0]
4 PIO[16]
5 GND
6 USB_XDM
7 USB_XDP
8 VCC3
9 RESET_B
10 RXD
11 TXD
12 CON_RD_B
13 CON_ALE
14 PSEN_B
15 PFLASH_CS_B
16 VCC
17 CON_WE0_B
18 CON_WE1_B
19 ADR_DAT_LO[0]
20 CAMIO[0]
21 CAMIO[1]
22 ADR_DAT_LO[1]
23 ADR_DAT_LO[2]
24 ADR_DAT_LO[3]
25 ADR_DAT_LO[4]
26 GND
27 ADR_DAT_LO[5]
28 ADR_DAT_LO[6]
29 ADR_DAT_LO[7]
30 ADR_HI[0]
31 ADR_HI[1]
32 ADR_HI[2]
33 ADR_HI[3]
34 VCC3
35 ADR_HI[4]
36 ADR_HI[5]
37 ADR_HI[6]
38 ADR_HI[7]
39 DA_DATA
40 AD_DATA
41 L3CLK
42 L3DATA
43 L3MODE
44 AUDBCK
45 AUDWS
46 CKO
47 DF_CLE
48 DF_ALE
49 DF_RD_B
50 DF_WE_B
51 FC_CS_B
52 DF_CS_B[1]
53 CAMIO[3]
54 VSYNC
55 HSYNC
56 PXCLK
57 PXD[9]
58 PXD[8]
59 PXD[7]
60 GND
61 VCC
62 PXD[6]
63 PXD[5]
64 PXD[4]
65 PXD[3]
66 PXD[2]
67 PXD[1]
68 PXD[0]
69 GND
70 DFAD[0]
71 DFAD[1]
72 DFAD[2]
73 DFAD[3]
74 DFAD[4]
75 AGND
76 XINW
77 XOUTW
78 XIN
79 XOUT
80 AVCC
81 DFAD[5]
82 DFAD[6]
83 DFAD[7]
84 LCD_E
85 LCD_RD_WE
86 LCD_RS_CS
87 SA[0]
88 SA[1]
89 SA[2]
90 SA[3]
91 SA[4]
92 SA[5]
93 SA[6]
94 SA[7]
95 SA[8]
96 VCC3
97 SA[9]
98 SA[10]
99 SA[11]
100 CAMIO[2]
101 DSP_SEL
102 GND
103 SA[12]
104 SA[13]
105 SA[14]
106 SD[0]
107 SD[1]
108 SD[2]
109 VCC
110 SD[3]
111 SD[4]
112 SD[5]
113 SD[6]
114 GND
115 SD[7]
116 SD[8]
117 SD[9]
118 SD[10]
119 SD[11]
120 SD[12]
121 SD[13]
122 SD[14]
123 SD[15]
124 MCLK
125 RAS_B
126 CAS_B
127 WE_B
128 CKE[0]
129 CS_B[0]
130 CKE[1]
131 CS_B[1]
132 GND
133 CKE[2]
134 CS_B[2]
135 CKE[3]
136 CS_B[3]
137 PIO[15]
138 PIO[14]
139 PIO[13]
140 GND
141 VCC
142 PIO[12]
143 PIO[11]
144 PIO[10]
145 PIO[9]
146 PIO[8]
147 PIO[7]
148 PIO[6]
149 VCC3
150 PIO[5]
151 PIO[4]
152 PIO[3]
153 PIO[2]
154 PIO[1]
155 PIO[0]
156 KEY_PAD[6]
157 KEY_PAD[5]
158 KEY_PAD[4]
159 KEY_PAD[3]
160 KEY_PAD[2]

2. Chip Architecture



Whole-Chip Block Diagram

The chip contains an embedded controller and an embedded DSP surrounded by several interface blocks. The controller will control these interfaces through the control interface block, which translates memory accesses by the controller to memory-mapped control/status register read/writes on the Control Bus. The controller will also control when and which routines the DSP will run.

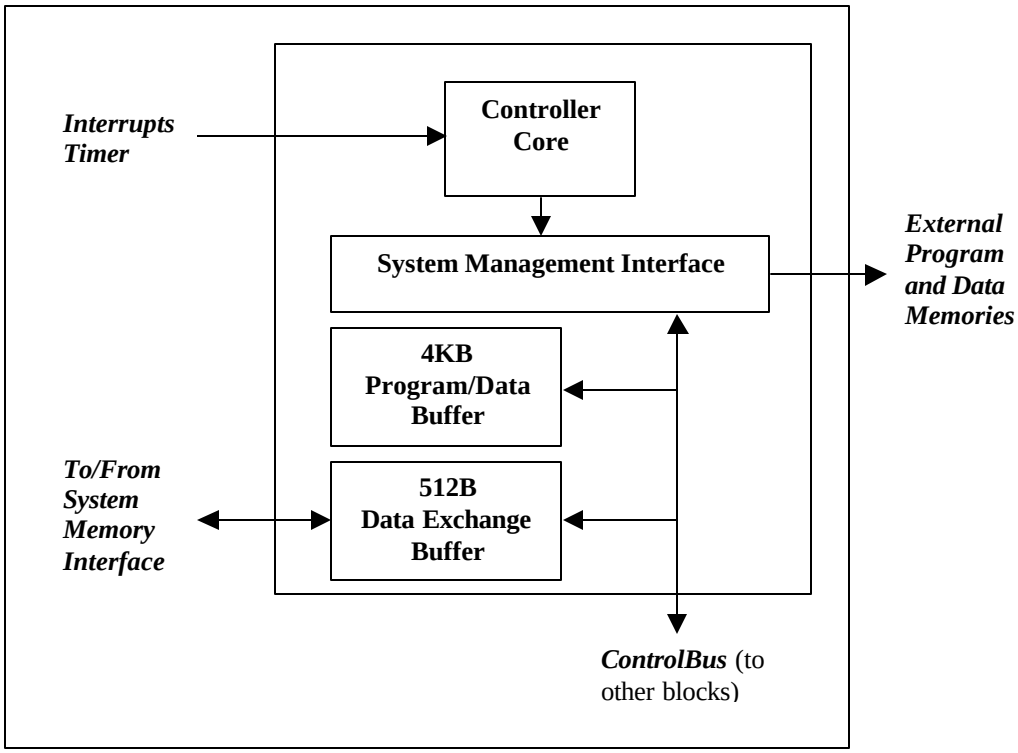
Most of the data will be stored in off-chip memory (in either SDRAM or SRAM). The system memory interface arbitrates between memory requests of the interface blocks as well as the controller and DSP buffers and serves the request with the most urgency.

The controller memory interface serves memory accesses of the controller. This block will either access off-chip EEPROM, on-chip SRAM, or memory mapped register, based on address. The controller and DSP buffers contain

double buffers for each memory block so that the DSP and controller can be running while the interface fetches next set of data from the system memory interface.

The external interface blocks include a CMOS sensor interface, a USB interface, an ADC/DAC interface, an RS232 serial interface, timers, auxiliary I/O, keypad support, and LCD driver. An R/F interface can be added with external glue logic to support wireless applications.

2.1 Controller Subsystem



Controller Subsystem Block Diagram

2.1.1 Controller Core

A controller is added to the DSP for two main benefits: 1) Control/application software can be written in high-level language; 2) Reduce the load on the DSP so it can concentrate on computation. Currently, the controller is an 8-bit micro-controller, but in the future a 32-bit processor can be used for greater flexibility and easier software porting. The controller is an 8051/8032 core.

2.1.2 Program Memory

The controller only can access 64KB of off-chip program ROM. In order to access more than the 64KB page limit, it can specify one of 8 pages (each page is 64KB, i.e. up to 512KB of program ROM) via software by one of several methods:

Method 1:

1) Software will write to a special page register
2) The page register will update the MSB of the program address, and then it will reset the controller.
3) The controller will start executing at the beginning of the new page.

Method 2:

1) Software will write to special page register
2) The page register will update the MSB of the program address, and then it will interrupt the controller via interrupt 1.
3) The controller will start executing at the location designated by the interrupt vector in the new page.
4) The software must clear the stack and the interrupt appropriately.

Method 3:

1) Software will call a special routine with the new page number and the desired location in the new page as parameters
2) The routine will write to the special page register
3) The page register will update the MSB of the program address
4) The routine continues to the next instruction, but this time its in the new page
5) The routine will jump to the desired location in the new page

*Note:

- Method 3 is supported by some 8051 linkers, which will link program bigger than 64KB this way, as long as the special routine is provided.
- For method 3, the special routine must be duplicated in every page, at the same location.

*Note: This is rather complicated, but it is necessary to get around the 64KB boundary. This needs the support of both hardware and software (although hardware is minimal), but it is flexible enough to accommodate all cases. Off-chip memory contains both program and data.

The program and data (including DSP code, which is treated as data by the controller)will most likely be contained in EEPROM or SRAM-like FLASH memory. Because these memories are typically slow, wait-states can be added to the external memory accesses. However, this limits the performance of the controller to about 5MIPS maximum. If more performance is needed, the program can be stored in SRAM, which can be accessed faster (i.e. with no wait-states at higher clock frequencies).

2.1.3 Data Memory

The original 8051 has an 8-bit data bus and 128 bytes of on-chip data RAM and up to 64KB of external data memory. The 128 bytes are still available in this controller. However, the 64KB of external data memory space is divided into several uses, including on-chip SRAM buffers, control registers for other functional blocks, DSP memory space, and external memories. The The System Management Interface will decoding the 16-bit external memory address and determine which is the appropriate source for the current access.

To begin with, there is a 4KB On-chip Program/Data Buffer. In normal operation mode, this 4KB buffer is used as data memory. However, there are two cases where this block can be turned into program memory (this is a special page):

1) During power-down mode, the controller can shut off access to external memory to save power, and run a simple program to update the clock and detect keys using a very slow clock and this piece of on-chip memory.

2) During program upgrade, the upgrade program is downloaded into this buffer, and then the program memory will be mapped to this buffer, and the data memory will map to a section of external program memory, and the upgrade can take place.

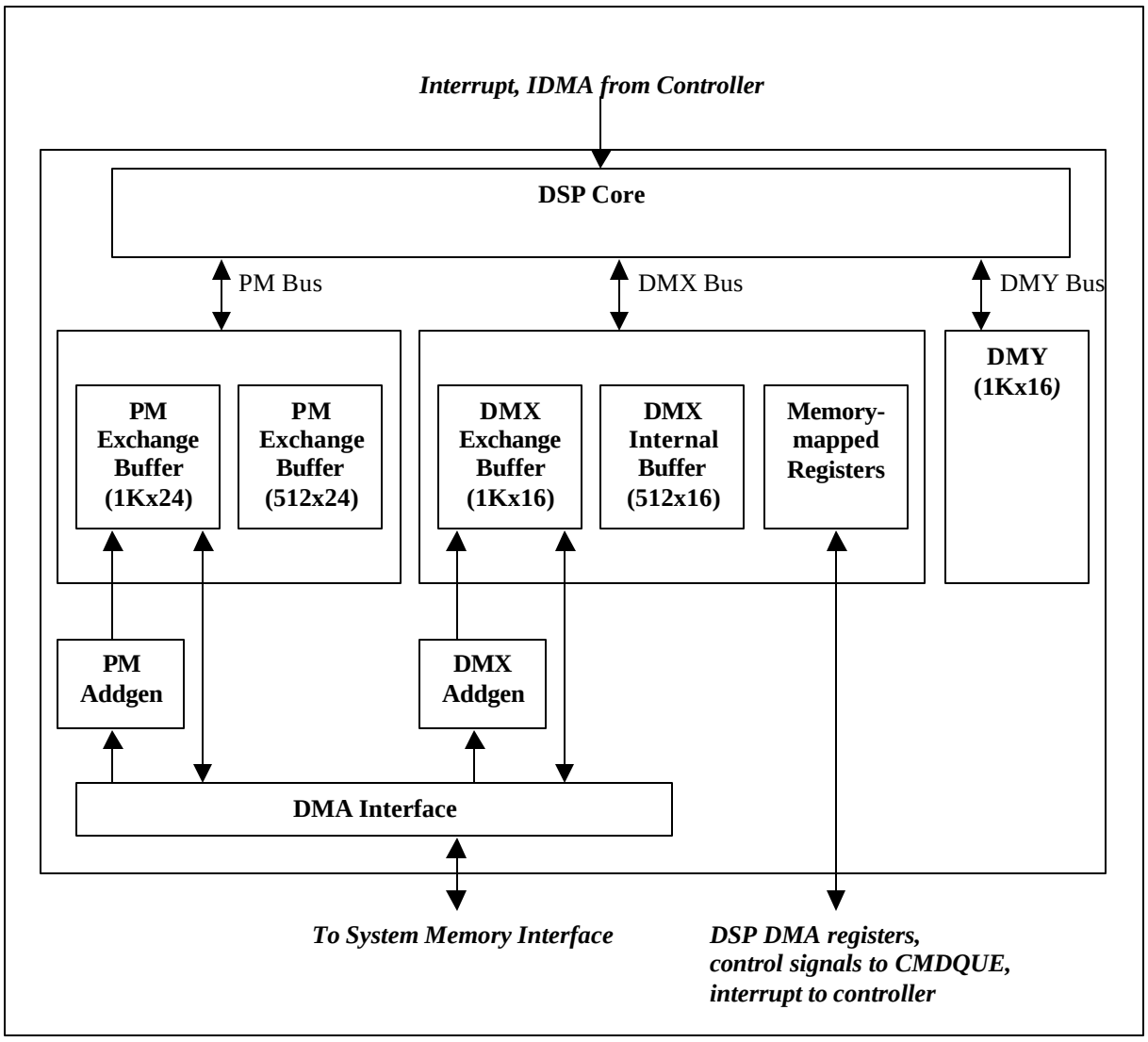
In addition, there is a 512 Byte buffer that can alternate between being accessible to the controller and the SDRAM interface. This is the way the controller can access SDRAM data.
The portion of the data memory space from 8000 to FFFF is reserved for control registers for the other functional blocks and DSP memory space.

Finally, one of the requirements is that the controller code can be upgraded , for example via USB. The controller can write to external program FLASH or EEPROM in 16KB banks by writing to external address 0-0x3FFF. In order to upgrade the entire program space, the 16KB bank is changed by writing to the data page register, which designate the higher bits of the address. The address space 0x4000-0x5FFF is reserved for an additional device that can be connected to this bus.

The external data memory space is as follows:

0000-3FFF	16KB of program upgrade memory
4000-5FFF	8KB of reserved memory (for glue logic or other devices)
6000-6FFF	4KB of controller memory (used by controller only for data and program).
7000-71FF	512Byte controller exchange buffer (used by SDRAM and controller to exchange data.
8000-BFFF	Register memory space
C000-FFFF	DSP Memory locations:
C000-C7FF	PM Internal Buffer
D000-DFFF	PM Exchange Buffer
E000-E3FF	DMX Internal Buffer
E800-EFFF	DMX Exchange Buffer
F000-F7FF	DMY

2.2 DSP Subsystem



DSP Subsystem Block Diagram

2.2.1 DSP Core

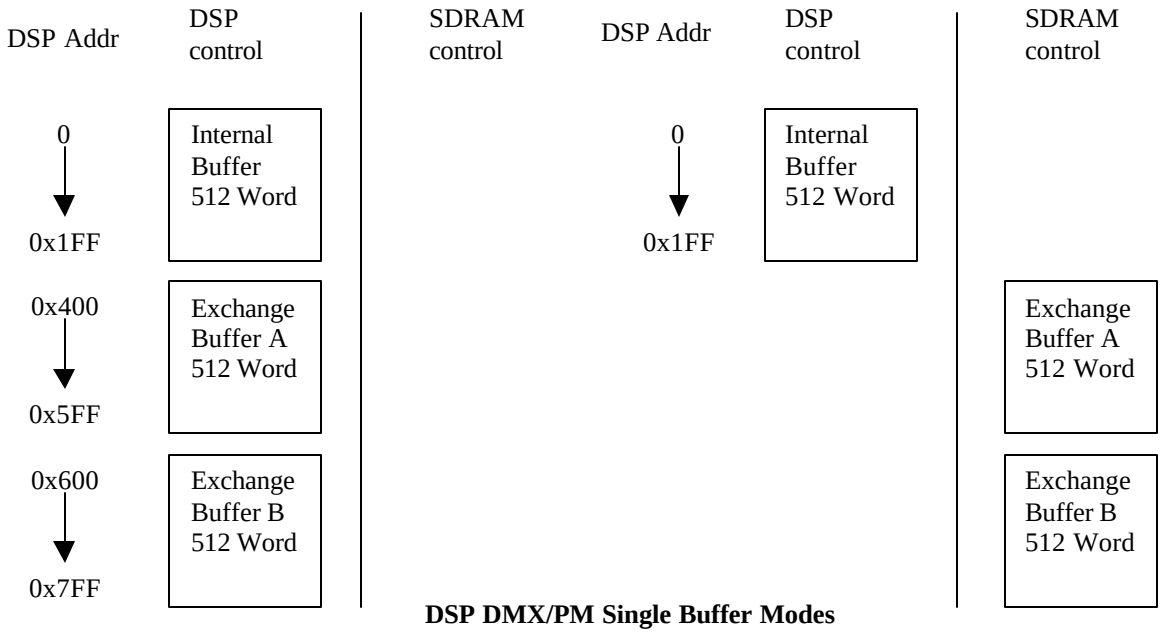
This block performs the fixed-point processing required for each application. The DSP is an ADI 2181 instruction-set compatible core with a single datapath. There are few differences that the DSP core has with AD2181:

1. In the instruction syntax, the "PM" maps to the DMY hardware
2. DIVQ and DIVS instructions are not implemented.
3. SLOW IDLE(n) instruction is not implemented, however IDLE instruction is
4. Two cycle delays on JUMP/CALL/RETURN/INTERRUPT instructions
5. Two cycle latencies for IFC Software Interrupt (one for ADSP 2181)
6. Two NOPs must be inserted under POPPCSTACK (one for ADSP 2181)
7. Two wasted cycles on DO/UNTIL instruction in the first loop pass
8. One NOP must be inserted after SEC_REG is enabled or disabled (none for ADSP 2181)
9. Avoiding JUMP or CALL inside 4th loop.

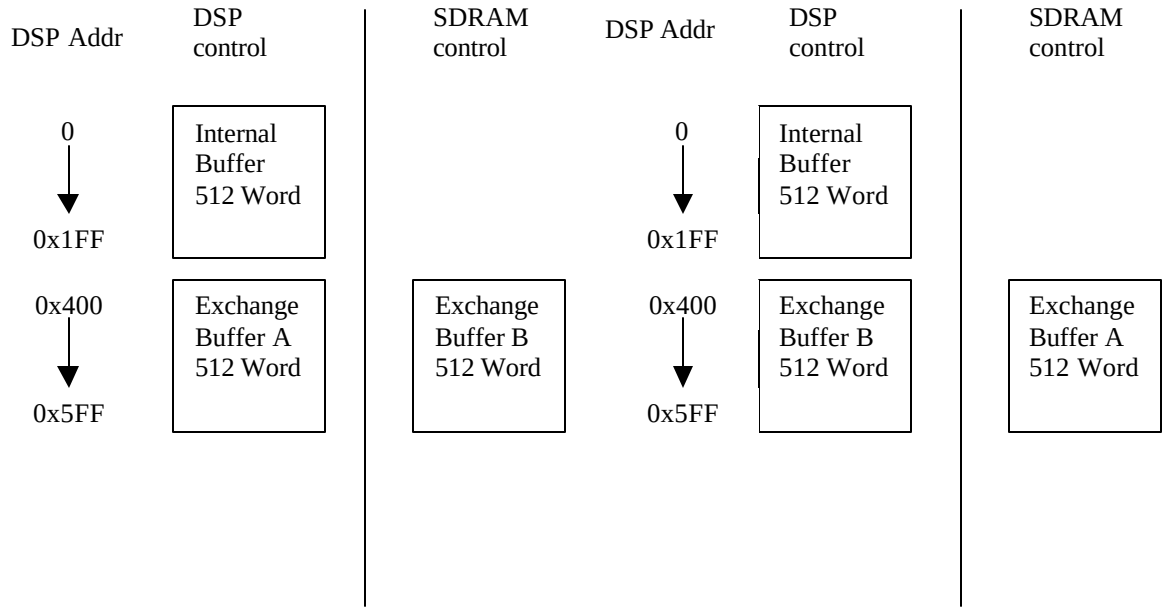
The controller controls the DSP execution. The controller can reset the DSP and boot the DSP through IDMA boot. The controller also specifies when and where the DSP should start, it controls the memory pages in external DSP memory (still on-chip, as described above), and it also knows when the DSP has finished running and puts into sleep mode. All of this is performed via the Command Queue, described below. The controller is allowed to interrupt the DSP, and the DSP can interrupt the controller.

2.2.2 DSP Memories

For program memory, the DSP have access to 0.5K word (24-bits) of program memory as well as 1K word buffer that can be used as a 1K word single-buffer or a 0.5K word double buffer (to allow the DSP to simultaneously execute the current routine while loading the next one). For data memory, the modified Harvard architecture in this chip supports two data memories: DMX and DMY, both of which is 16-bits wide. DMY is 1K word (1Kx16bits), and should contain mostly coefficients, constants, or other values that do not change frequently. It is accessible by the DSP only. DMX contains a 0.5K word piece of memory as well as a 1K word buffer that can be used as a 1K word single buffer or as a 0.5K word double buffer. If the buffer is used as double buffer, then it is possible to have the DSP use one buffer and load/store data via DMA in the other. Otherwise, in single-buffer mode, the buffer can either be used by the DSP only (no DMA can occur) or by DMA only (the DSP cannot access). The controller can access all memory locations of the DSP (including PM, DMX, DMY) via the IDMA.



DSP DMX/PM Single Buffer Modes



DSP DMX/PM Double Buffer Modes

2.2.3 DSP Memory Mapped Registers

An additional mode has been added to the DSP to allow it to directly exchange data with the SDRAM via DMA instead of having to go through the controller. This is useful since the DSP can directly control the data flow, and also it is much more efficient since the DSP is an order of magnitude faster than the controller. The registers are written via writing to DMX address between 0x200-0x280. This region is not occupied by actual SRAM.

2.3 CmdQueue

The command queue logic allows communication between the controller and the DSP core. It contains a command FIFO (8 deep) which can allow the controller to write a series of command for the DSP to perform. Each command contains one bit which determines whether that command needs to wait for the previous commands to finish or can start immediately. The commands include:

Bit 15:12 Command		Bit 11:0
1xxx	Wait for previous commands to finish	
x000	Start DSP execution	PC
x100	Load DMX Exchange Buffer	Address
x101	Store DMX Exchange Buffer	Address
x110	Load PM Exchange Buffer	Address
x111	Store PM Exchange Buffer	Address
x001	Swap DMX/PM Double Buffer	DM or PM. Swap single or double buffer
x010	Reset DSP	

Once the controller issues a series of commands to the cmdqueue, it can go off and do other things. The cmdqueue will decide which commands can be executed at a particular time (based on the MSB of the command). For memory load/store instructions, the cmdqueue will interact with the SRAM controller to begin the data transfer. Once that is finished, the SRAM controller will issue assert a flag. For the DSP execution commands, the command queue will

reset the DSP, insert a new PC through IDMA, and start DSP execution. Once the DSP routine is finished, the last commands will write to DM 0x3FF, which triggers a special flag for the command queue. The command queue will decide based on flags from the SRAM controller and the DSP the progress of the FIFO and whether to pop a entry or not. (Note: the wait bit means that this command must wait for ALL of the previous commands to finish).

The controller can monitor the number of entries in the cmdqueue, or it can be interrupted when the command queue contains a certain number of commands (this can be programmable via control register). For example, an interrupt can be programmed when there's one entry or zero entries in the command FIFO (in most cases zero will be used). If more than 7 commands are written to the command queue, an overflow flag will be set and any additional commands written to the command queue will be ignored.

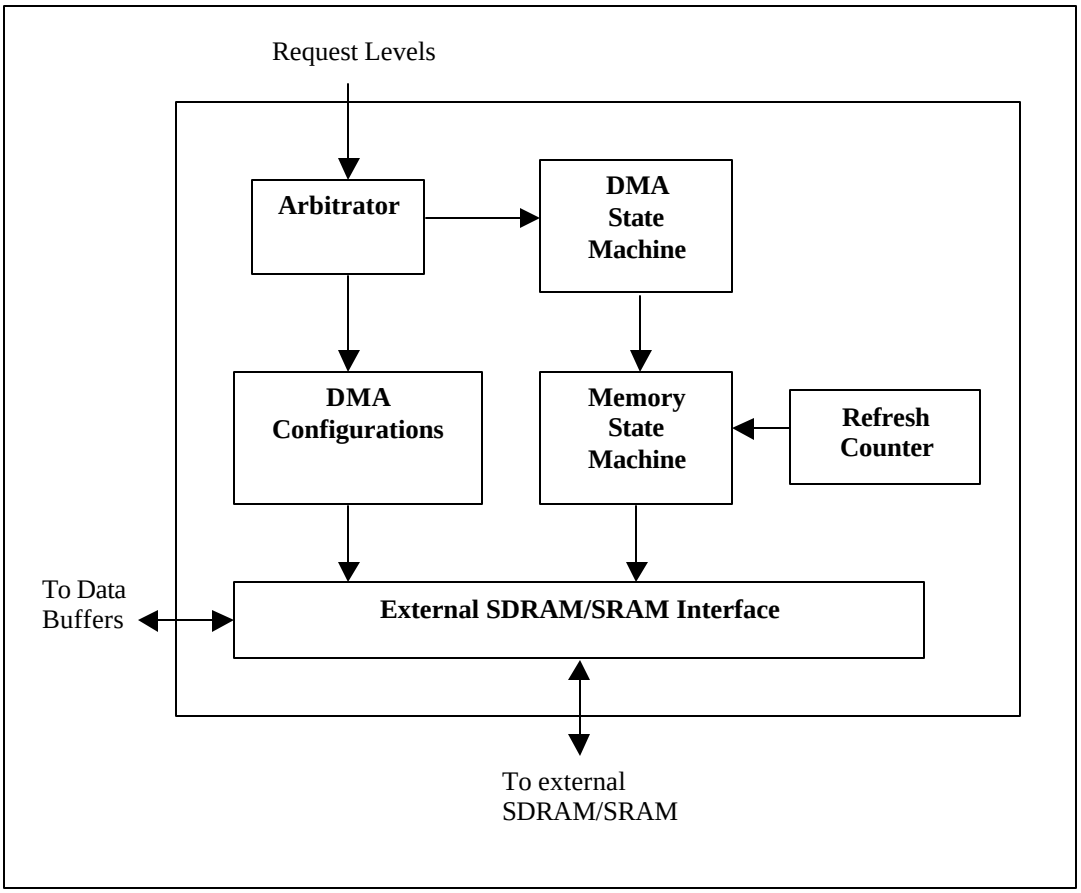
When the execute instruction is performed, the DSP does the following two things:
1) It inserts a branch instruction, with the branch location designated by the CMDFIFO entry, into a specific location in PM.
2) It then causes an interrupt in the DSP.

With the combination of these two actions, there are many ways to start DSP execution at a desired entry point. One such procedure for starting DSP execution by the CMDFIFO:

1) DSP is in IDLE mode after executing some initial instructions.
2) The command queue uses the IDMA to insert a branch instruction to a designated location (usually the interrupt vector location), with the branch target as the desired starting point for execution.
3) The command queue causes an interrupt to the DSP
4) The interrupt routine is executed.
5) The DSP branches to the desired routine.
6) After execution is done, the last instruction in the routine is an RTI
7) The instruction after the IDLE instruction will jump back to the IDLE instruction.
8) The DSP is back in IDLE mode.

Finally, the controller must grant permission in order to allow the DSP to directly set up its own DMA (see DSP section and DMA section). This is done by setting a special register. In order to avoid conflict, no commands in the command queue will be executed while the DSP has control of the DMA channels. The programmer can still write commands to the command queue, but they will not execute as if all of them had a WAIT condition. The command queue will resume normal operation once the controller disables DSP's access to DMA.

2.4 System Memory Interface



System Memory Interface Block Diagram

2.4.1 External Memory Support

This blocks interfaces with external system memory, either SDRAM or SRAM. The external data bus can be either 16-bit or 8-bit, but the internal bus is 32-bit. For SDRAM, there are 15 address bits and 4 chip selects which can address up to 4 pieces of 16Mx16 SDRAM, for a maximum of 64MB. For SRAM, there are 20 address bits and two chip selects, which can address up to 2 pieces of 1Mx16 SRAM. Note however, that SRAM of this size is not widely available, and we generally reserve the 2nd chip select for glue-logic for RF interface.

2.4.2 DMA Channel Arbitration

The System Memory Interface serves multiple blocks and arbitrates between the different blocks that need to access external memory. Each block has a DMA channel with a two bit request level. In addition, they all have an inherent priority, so that when two channels have the same request level, the channel with the higher priority will win. The priority level can be programmed by controller or determined by FIFO fullness, depending on the channel. The following is a list of channels:

Priority	Channel	Direction	Request Method
1 (highest)	Handspring/Dataflash out	From Memory	FIFO fullness
2	Handspring/Dataflash in	To Memory	FIFO fullness
3	Camera	To Memory	FIFO fullness

4	Codec channel 1		From Memory	FIFO fullness
5	Reserved			
6	Codec channel 0/USB out		From Memory	FIFO fullness
7	Codec/USB in		To Memory	FIFO fullness
8	SDRAM Refresh		Timer control	
9-10	Controller	Bidir	Controller set	
11-12	DSP PM	Bidir	Controller set	
13-14	DSP DM	Bidir	Controller set	
15-16	Reserved			

Arbitration is a 3 step process. First, requests for each level is determined (from the requests and whether a DMA is set up for that channel). Then each level goes to a priority encode to come out with whether there is a request for that channel and a channel to service. Finally, the channel for the highest valid request will be serviced.

When the controller detects a request with a higher arbitration priority (based on the above protocol) than the current one, it will interrupt the current DMA, store the location of the last access (i.e. where the stop occurs), and the proceed to service the new request. The interrupted request can resume at where it stopped after its arbitration priority once again becomes the highest. This scheme can be repeated for all the channels active at the same time.

There should be control registers which set a minimum transfer number to avoid overhead with constantly switching channels. All of our channels require relatively low bandwidth, and they are rarely all in use, so I don't think there will be much switching back and forth between channels which will deplete the bandwidth.

Finally, there needs to be an automatic refresh channel with a timer which will interrupt the controller and refresh the memory at the appropriate times.

2.4.3 External Memory Interfaces

The arbitration block contains all the arbitration logic and passes on a DMA identifier to the DMA Configuration block and the DMA State Machine block. The state machine keeps track of the current state of the DMA being served and determines whether an interrupt occurs. The DMA Configuration block contains the control registers which define a particular DMA channel (as described above). The Memory State Machine contains the state necessary to interface with the external memory types (currently 8-bit and 16-bit SDRAM and SRAM are supported), and the External SDRAM/SRAM interface interfaces with the SDRAM/SRAM chip with the appropriate control signals, address, and data. There is also a refresh counter to automatically refresh the SDRAM.

The SRAM and SDRAM pins are shared, so either SRAM or SDRAM can be used as external system memory, but not both.

2.4.4 DMA Setup

The controller must set up a DMA for each channel before requests for that channel is arbitrated. The DMA is set up by specifying the following:

- Starting address
- Number of Consecutive transfers per "line" (X)
- Number of "lines" to transfer(Y)
- and, if necessary, the space between adjacent lines (i.e. how much to add at the end of one "line to the beginning of the next.(width).
- Note: The starting address is a BYTE address (although the 2 LSB are ignored). Other than that, all the other values use 32-bit DWORD as units. This is because the internal bus width is 32-bits.

For details about DMA setup, please refer to the relative chapter in Programming Guide.

2.4.5 BYTE access mode

Because of the 32-bit internal bus, it is difficult to obtain data that crosses the DWORD boundary. There is a method for reading data starting at non-DWORD-boundary locations. Note, however, that this is for read only (there is no provision for writing to non-DWORD-boundary locations), and the number of bytes to read per lines is still multiples of 4.

With the above caveats, this is how the BYTE access mode works. The controller will designate which BYTE to start from. If it's 0, then BYTE mode is disabled and the DMA works as normal. If it's 1, for example, then for each DMA for the specified channel (only one channel can have BYTE mode at a time) will read a DWORD and store the bottom 24 bits into a storage register. The DMA will then read another DWORD and piece together the 24-bits stored previously and the top 8-bits read this time into a 32-bit data to be written to the requesting channel. It will again store the bottom 24-bits for next time. This operation mode does not send any data from the first read to the requesting channel. It only saves the bottom bits for next time, and throws away the top bits. This is called the deduct mode. In the non-deduct mode, the top bits of the first read is pieced together with whatever was stored in the storage registers from last time for the first 32-bit data. The non-deduct mode is useful for continuing a BYTE access read at a different location. The following table list the BYTE mode results in the deduct mode (each letter represent one BYTE, and the left side is the MSB).

Data in Memory	BYTE_SEL = 00	BYTE_SEL = 01	BYTE_SEL = 10	BYTE_SEL = 11
ABCD	ABCD	BCDE	CDEF	DEFG
EFGH	EFGH	FGHI	GHIJ	HIJK
IJKL	IJKL	JKLM	KLMN	LMNO
MNOP	MNOP	NOPQ	OPQR	PQRS
…	…	…	…	…

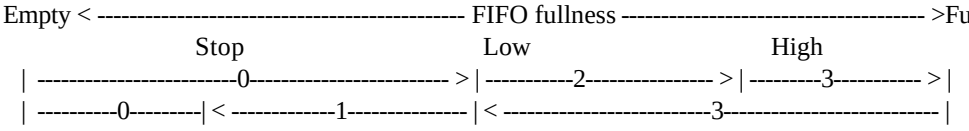
2.4.6 FIFO's and interface to SDRAM controller

FIFO's are blocks of SRAM with a read pointer chasing a write pointer (one may never pass the other). The fullness of the FIFO is determined by the difference of the two pointers. Since most of the FIFO's in this chip are bidirectional, one must be very careful with both FIFO overflow and underflow in both situations. Make sure that the request levels to the SDRAM controller is set properly so that it can start/stop in time. The size of the FIFO's are application dependent, but since we have all low bandwidth channels, it's probably good enough to set one size (i.e. 64x32) for all channels to reduce design time. The FIFO must provide status flags and interrupts to the controller if there is an underflow or overflow.

The FIFO must interface with both a data producing/needing block and the SDRAM controller. I will first discuss the SDRAM controller. A FIFO sends a 2-bit request level to the SDRAM controller. If that FIFO is serviced, the SDRAM controller will send it a data valid signal. On the following cycle, the FIFO will either write the 32-bit data onto the databus if it's writing to SDRAM or read the 32-bit data from the databus if it's reading from SDRAM.

The request level is determined by two sets of registers (one for reading and one for writing) setting three checkpoints (stop, low, high) which will trigger different request levels (note there is some hystersis so channels won't keep switching between request and no request):

FIFO write to SDRAM:

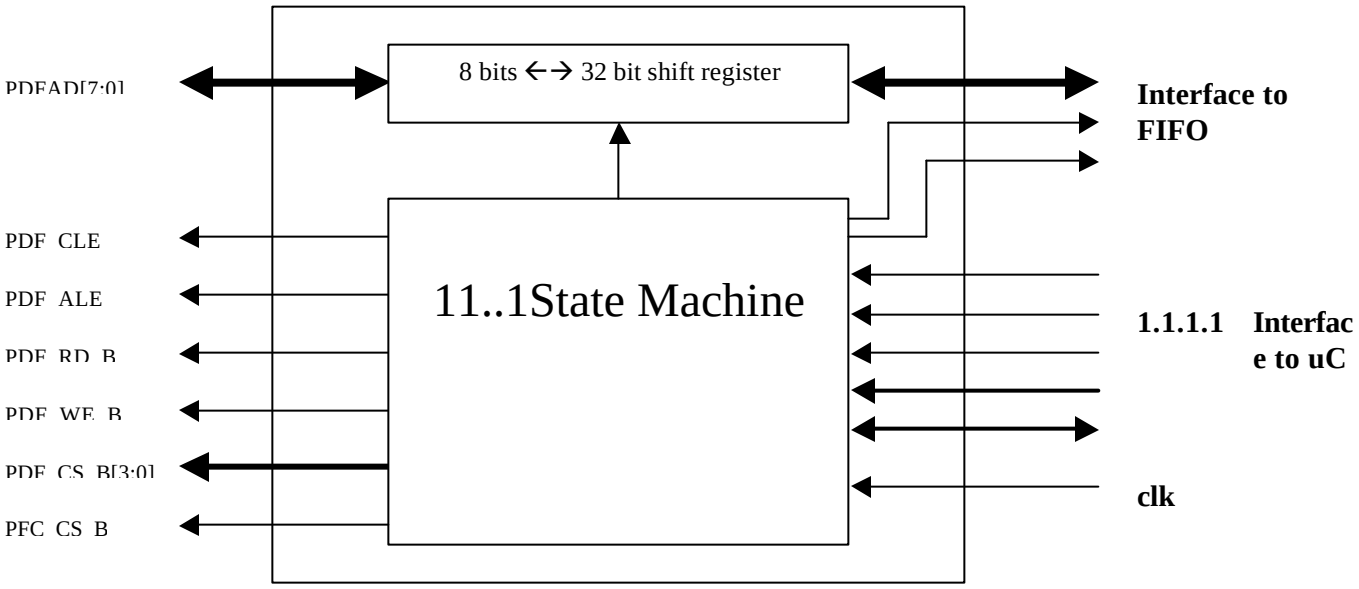


FIFO read from SDRAM:

In situations where the block and the SDRAM controller is trying to access the FIFO at the same time, delay the write until the next cycle. Since the databus is 32-bits and the SDRAM is 16-bits, and all of our blocks are low bandwidth, there is never a case with two consecutive writes or two consecutive reads. If there is a conflict in one cycle, then the next cycle is definitely free, so the write can be performed then.

It is necessary to include a flush-bit to the input FIFO's so that the last few words of the inputs will not be stuck in the FIFO at the end of a data block. When the flush bit is set the request level is 1 even below the low start level. This insures that the FIFO is emptied even if new data is not coming in.

2.5 Data Flash Interface



Data Flash Interface Block Diagram

This interface provides the connection to the smart media card and smart flash through an 8bit bus and corresponding control signals.

The PDF_CLE and PDF_ALE indicate the bus, PDFAD[7:0] which is shared with the LCD interface, is in command cycle, address cycle or data cycle. PDF_RD_B and PDF_WE_B correspond to the writing and reading operations, respectively. Totally 5 chip-select are provided. PFC_CS_B is exclusively for smart media card. And the PDF_CS_B[3:0] are for on board smart flash. The address space of each smart flash is independent to each other. One thing needed to be pointed out is that the PDF_CS_B[3:2] are shared with the chip select of SDRAM, PSDRAM_CS_B[3:2]. So, totally 6 chip select are provided for on board smart flash and SDRAM, and they can be configed as “4 smart flash + 2 SDRAM” or “2 smart flash + 4 SDRAM”.

Two modes, the DMA mode and uC mode, are provided for the data transferring. In the DMA mode, a DMA channel between the smart flash and SDRAM is created through the shift register and FIFO. The bandwidth of DMA mode is not lower than 1.2Mbytes/s. In the uC mode, the uC can read or write the smart flash directly. The bandwidth of uC mode depends on the reading and writing external register speed of the uC.

Finally, the data flash interface can go to sleep mode by stopping its clock.

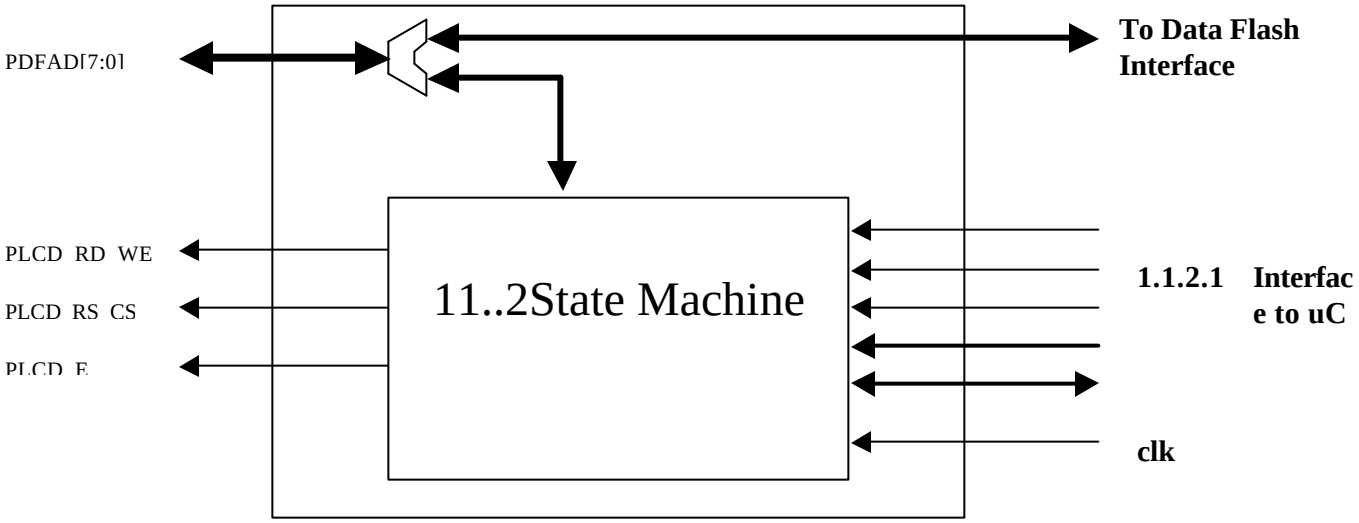
2.6 LCD Interface

LCD interface provides the connection to SAMSUNG KS0073 LCD controller or its compatible products. This interface work in 8-bit mode, and the data bus is shared with that of the data flash. The signal PLCD_RS_CS indicated current operation is for instruction register or data register of the LCD controller. The signal PLCD_RD_WE indicates the reading or writing cycle. PLCD_E is the reading and writing enable signal.

Only the uC can read or write the LCD controller. In order to read or write the LCD controller’s instruction register or data register, the uC only need to read or write the some external registers, which locate in the state machine. The state machine will transfer the uC’s reading or writing operation to corresponding control signal.

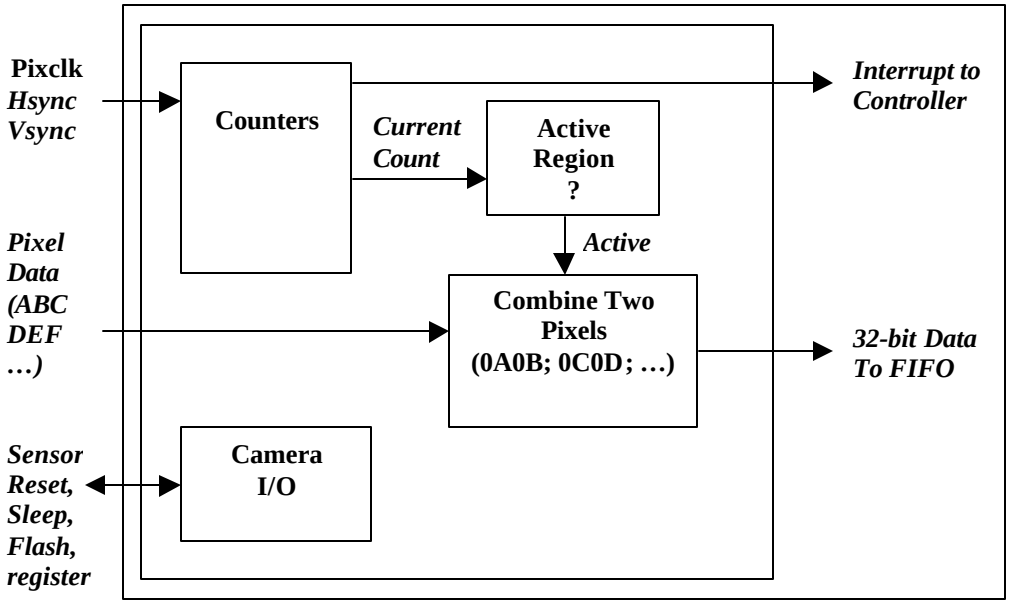
The bandwidth of data transferring between LCD controller and Palm1 depends on the reading and writing external register speed of uC.

The LCD interface can go to sleep by stopping its clock.



LCD Interface Block Diagram

2.7 Camera Interface



Camera Interface Block Diagram

This block interfaces with the CMOS imaging chip and processes the 10-bit input from the CMOS sensor through a series of hardwired logic. The image is framed by HSYNC and VSYNC signals and is clocked by PIXCLK. The input interface is slave mode only, and it can invert the sync signals and pixel clock as needed. Two counters will count the X and Y positions, respectively, of the current pixel. The user can define an active region by specifying the beginning and ending X and Y coordinates. In the active region, each 10-bit sample is zero-filled to a 16-bit value. Two 16-bit values are pieced together and sent to the FIFO to be DMA'ed to the system memory.

The camera interface also contains for I/O signals which can be used as a serial interface to program the sensor's internal register, reset/power-down the sensor, and/or control the Camera Flash.

Finally, the sensor can generate an interrupt to the controller at a specified line during each frame. This may be useful for video interrupts.

2.8 Codec Interface

Two types of codecs are supported by the chip – AC97 codec and Philips UDA codec. These two types of codec have very different interfaces, but they share resources such as pins and FIFO. One one codec is supported at a time.

2.8.1 AC97

The AC97 interface is a serial interface (with bitclk, frame_sync, AD data and DA data) which follows the AC97 frame format. Bitclk and AD data are inputs from the codec, and frame_sync and DA data are outputs to the codec. This interface also allows the controller to read and write registers in the codec as part of the AC97 frame. It also has the ability to turn on/off the left and right sound channels separately.

2.8.2 UDA

The UDA interface is also a serial interface with bitclk, frame_sync, AD data and DA data. All four of these pins are shared with the AC97 codec. In this case, however, only AD data is an input from codec. The bitclk, frame_sync, and DA data are all outputs to the codec.

The UDA interface supports two formats: I2S and LSB first.

The controller can determine the frequency of the bitclk and the frame_sync. It can also turn on/off the left and right sound channels separately.

The UDA interface also supports an additional L3 serial interface, which can be used to read/write control registers in the codec. This interface requires an additional 3 pins, which can be used as general purpose I/O when not required by the codec.

2.8.3 Codec FIFO

The two codec interfaces not only share pins, but they also share the same FIFO's. Each codec expects to receive a 32-bit value from the FIFO which contains two samples. If both stereo channels are active, then these two samples would be one left-channel sample and one right-channel sample. If only one channel is active, then these would be two consecutive samples.

The FIFO can operate in two different modes. The first is all the data is DMA'ed from one set of data using one FIFO. This is useful for mono-mode, where there is only one set of data, or in stereo mode where the data is already interlaced into LR LR LR samples.

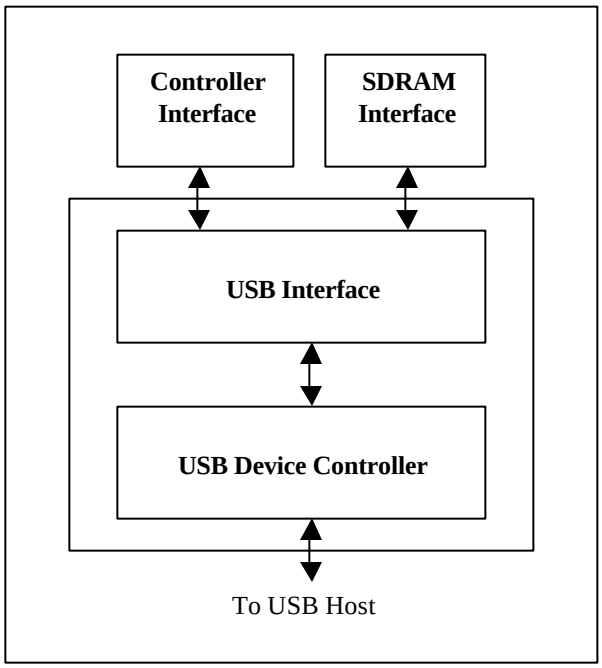
The other mode is using two FIFO's to DMA two different sets of data and interlace them before sending the output to the codecs. This is useful for stereo-mode applications where the left and right channel data are placed in separate locations (i.e. MP3).

For AD input, only one FIFO can be used, so only the first mode of operation is supported.

2.9 USB Interface

USB block includes a high-speed USB Device Controller and a USB interface to controller and SDRAM. The USB interface provides both MCU I/O access and DMA transfer mode.

The USB Device Controller has built-in USB transceiver, Serial Interface Engine (SIE) and command decoder (UBL) to be fully compliant with high-speed USB specifications revision 1.1 and 1.0 at operation rate up to 12MHz. It has a configurable 128-byte Ping-Pong buffer to optimize the DMA transfer throughput. It supports four user-configurable USB end-point control units.



USB Interface Block Diagram

In MCU access mode, controller can access the USB Device Controller through 4 register ports.

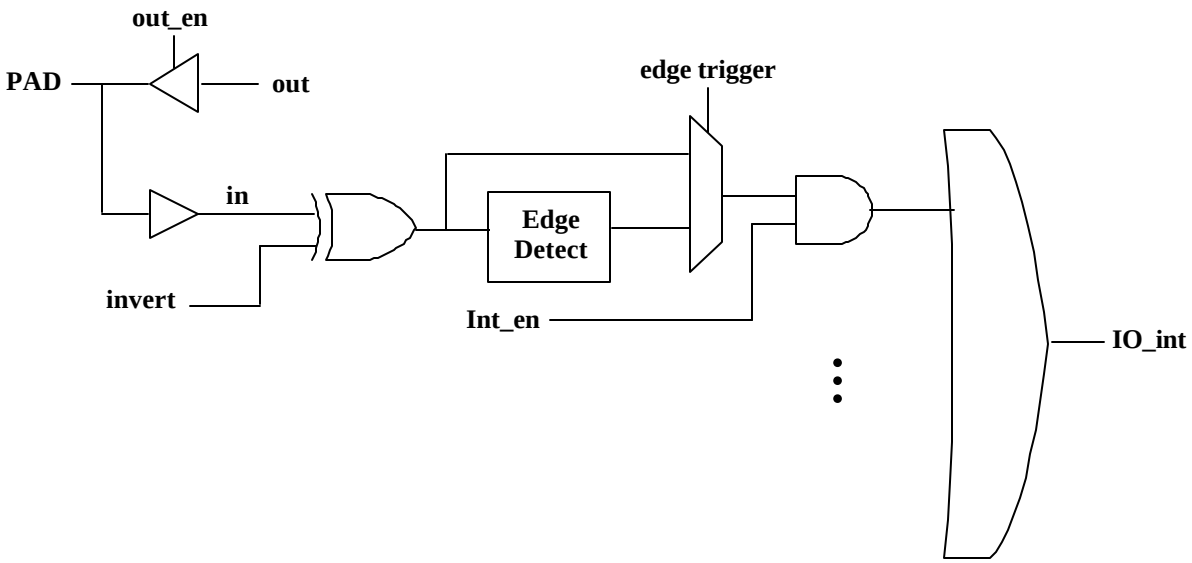
In DMA transfer mode, the USB interface directly transfers the data between the SDRAM interface and the USB Device Controller. In DMA transfer mode, MCU can no longer access the USB Device Controller unless it sets a register bit to disable the DMA. Once the DMA is disabled, it will be suspended until it is enabled again.

The MCU starts the DMA transfer by setting the DMA enable bit. Before that, the MCU should also initialize a DMA counter with the number of bytes to be transferred. The DMA counter counts down when the DMA is going. During the DMA transfer, MCU can probe the counter's value to see how far the DMA is going. When the counter counts to zero, the DMA stopped and is disabled automatically.
Due to the reason that the bandwidth of USB is far lower than the DMA, it's needed to add wait states. Currently we can add up to 16 wait states.

The USB Device Controller can also sent interrupt request to the MCU directly if it's enabled. To enable the USB interrupt, not only the interrupt enable register should be set properly, but also one of the internal register of the USB Device Controller should be programmed – Interrupt Enable Register (INTENR). When interrupts occurs, the MCU can judge which interrupt it is by reading the Interrupt Flag Regis ter (INTFLR). Please note that the interrupt flag bit has no meaning unless that interrupt is enabled.

2.10 Programmable I/O, Keypad, Interrupts and Timer

2.10.1 Programmable I/O



I/O Structure

This chip contains 17 programmable I/O, each of which has a structure as pictured above. Each I/O can be programmed as input or output (**out_en**), and the output value can be programmed to either 1 or 0 (**out**). The input value (**in**) can be read directly or can generate an interrupt. That interrupt can be programmed to be active high/low (**invert**), edge/level triggered (**edge triggered**), or masked (**int_en**). The interrupts from all the I/O bits is OR'ed together to generate the signal **IO_int**.

2.10.2 Keypad

The keypad pins have exactly the same structure as above for the programmable I/O's. These are treated separately only so it is easier to distinguish a keypad interrupt from a general I/O interrupt. There are seven keypad pins (4 column pins are pull up, and 3 row pins are pull down). This means that 12 keys can be supported directly. If more pins are desired, then a encoder must be added externally.

One method for keypad operation is as follows:

1) The controller drives a 1 out of the column pins and configures the row pins as inputs (which reads a 0 since they are pull-down).
2) The controller enables active high interrupt for the row pins.
3) If one of the keys is pressed, one of the row pins will be connected to a column pin and becomes a 1.
4) The controller is interrupted
5) The controller determines this is a keypad interrupt, and then determines the row of the key based on which of the row pins reads as a 1.
6) The controller then changes the column pins to inputs and drives 0 out of the row pins.
7) All of the column pins will read 1, except one which is connected to a row pin.
8) The controller will determine the column based on which column pin reads as a 0.
9) The controller now can determine the key based on the row/column, and takes appropriate action for that key.

2.10.3 Interrupts

All of the interrupts for this chip is mapped into int0 of the controller interrupt. Once interrupted, the controller must determine the source of the interrupt by reading the interrupt status registers. Each of the interrupts have a invert, level/edge select, and mask bit similar to the I/O interrupt described above. The only exceptions are keypad and I/O, which have multiple bits are thus is treated separately at a level below the top interrupt level. The interrupt sources are:

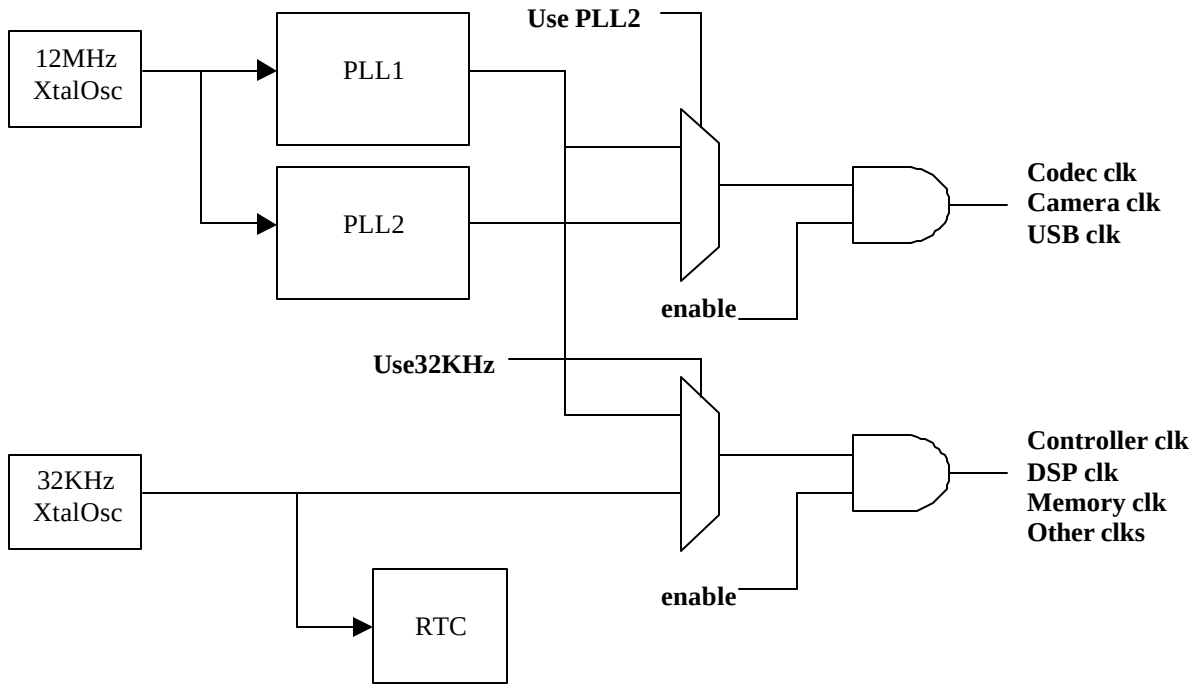
I/O	Externally generated interrupt
Keypad	Keypad interrupt
DSP	DSP interrupt controller
DMA	Specified DMA is finished
Cmdq	Command Queue contains specified number of entries
Camera	Sensor is a specified location in image
USB	USB block interrupt
Handsp	Handspring module interrupt

Int1 of the controller interrupt is reserved for program paging switching (see above).

2.10.4 Timer

The controller contains an 8-bit loadable event counter. A 16-bit programmable counter is added to augment it. The output of the 16-bit counter is sent to the controller as the event. The counter is driven by an external crystal, which should typically be a 32KHz watch crystal. By enabling the timer interrupt in the controller, it can be interrupted at a regular interval, for real-time clock, etc.

2.11 Clocks



Because PALM-I is very power-conscientious, the clock structure built to minimize power consumption for various applications.

PALM-I has two crystal oscillators: one at 12MHz for driving the PLL's, and one at 32KHz for the real-time clock and also to be used as system clock during power-down mode.

PALM-I also has two PLL's, each takes the output of the 12MHz oscillator as the input clock and has a locking range of 5-50MHz. These PLL's can be programmed to different frequencies depending on the exact requirement of different applications. The PLL's can also be programmed to bypass the 12MHz clock and put into sleep mode for power saving. The reason for two PLL's is so that clocks of certain peripherals, such as codec, camera, and USB, may be asynchronous to the system clock for some applications. If the signal **Use PLL2** is asserted, then the clocks for codec, camera, and USB are from PLL2 and are asynchronous to the system clock. These blocks are capable of supporting asynchronous interfaces.

To further reduce power, the clock signal going into each functional block can be disabled. Since all of our logic is standard-cell, this reduces the power consumption of certain functional blocks to 0.

To switch clock frequencies, the following steps must be taken:

1) Put the PLL into bypass mode (clock is now 12MHz)
2) Set the up/down dividers for the PLL as well as the compensation values
3) Wait until the lock signal is asserted
4) Switch to the new frequency by disabling bypass mode.

Note: Do not change the frequency of the PLL without putting it into bypass mode first. That could cause errors due to a unstable clock. You must wait until the locking signal is asserted before disabling bypass.

3. Programming Guide

3.1 DMA

3.1.1 Overview

The PALM-I chip will have most of the storage in off-chip "System Memory." This is usually SDRAM (up to 64MB is supported), although up to 2MB of SRAM can be used in place of SDRAM.

To access this memory, DMA channels are provided for transfer from on-chip memory blocks and FIFO's to/from this "System Memory". There are totally 16 DMA channels (listed below), and them must be programmed in order for data to be exchanged between the System Memory and the PALM-I chip. This document will describe how to setup different types of DMA transfers for different functional blocks. For the exact register definitions, please see the document palm1.ers.

1. Handspring/Dataflash out
2. Handspring/Dataflash in
3. Camera in
4. Codec1 out
5. (Reserved)
6. Codec0/USB out
7. Codec/USB in
8. SDRAM Refresh
9. Controller buffer
10. (Reserved)
11. DSP PM
12. (Reserved)
13. DSP DM
14. (Reserved)
15. (Reserved)
16. (Reserved)

There are three DMA channels directly to SRAM blocks with the chip (i.e. Controller buffer, DSP PM, and DSP DM), and the rest are DMA channels to FIFO's, which can be shared between functional blocks.

3.1.2 DMA Basics

In order to define a DMA, the programmer needs to know a few things:

1) Is this a read from memory or write to memory?
2) Where to get the data?
3) Where to put the data?
4) How to get the data (i.e. consecutively, or with jumps in between)?
5) What is the priority of this transfer?

These parameters are defined by a series of registers which the programmer must set in order to setup a DMA properly. Each of the channels listed above has its own registers, which means that DMA's for different channels can co-exist at the same time. Each set of the registers define the following values:
(Note: registers for DMA channel 1 is used as an example here. All the channels have the same register definitions)

- Starting Address – This is a 25-bit value which defines the starting address of the DMA transfer in System Memory (i.e. either SDRAM or SRAM). If SRAM is used, then the top bits are ignored. The starting address is

defined in four registers: *DMA1_START_ADDR1, DMA1_START_ADDR2, DMA1_START_ADDR3,* and the top bit is shared with other values in *DMA1_HIGH*. (Note: *DMA1_START_ADDR1* will trigger the start of a DMA and must be set AFTER all the other registers). The Starting Address is a BYTE address, although it must start on a DWORD boundary (i.e. the bottom two bits must be 00).

- X value – This a 10-bit value which defines how many consecutive DWORD's to access per line of DMA. The bottom 8-bits of this value is in the register *DMA1_X1*, and the top two bits are in the register *DMA1_HIGH*.
- Y value – This a 10-bit value which defines how many lines of DMA to perform. The actual lines of DMA is Y+1. For example, to DMA one line, Y = 0; to DMA 10 lines, Y = 9. The bottom 8-bits of this value is in register *DMA1_Y1*, and the top two bits are in the register *DMA1_HIGH*.
- Width – This is a 10-bit value which specifies the spacing between two lines (in DWORDs). For example, if the width = 100, X = 10, and Y = 5, then each time the DMA gets 10 DWORD's, it will jump 90 DWORD's to reach the start of the next line. Since width registers are often fixed for a particular application, the PALM-I provides a set of 4 width values (*SDRAM_WID0L, SDRAM_WID0H,* etc.) that can be pre-initialized. A particular DMA only needs to specify which of the 4 width register to use in the *DMA1_HIGH* register.

For DMA to SRAM blocks (i.e. DSP PM/DM, Controller Buffer), the direction of the DMA must be specified. In addition, an address for the start of the DMA in that memory block must be specified. Finally, a request level (1, 2, or 3, with 3 being most urgent) must be specified for that channel. These are done differently for the different buffers and will be discussed more in detail later.

For DMA to FIFO's, the direction of the DMA is fixed, and the starting address in not applicable. The request level of a DMA channel is dynamically varying depending on how full/empty the FIFO is. For example, if it's a DMA to System Memory, and the FIFO is also full, then the request level is high. On the other hand, if the DMA is the other direction, then the request level is low. The request level is determined by three water marks – high (high request), low (low request), stop (stop requesting) – which are programmed for each FIFO channel. Also note that FIFO channels are shared between the CODEC and USB, and between Handspring and Data FLA SH. These sharing is determined by registers *CODEC_SHARE* and *DF_HS_LCD_MUX*, respectively.

Note: Each DMA channel only has one set of configuration registers, which changes values while the channel is being in use. Thus, the programmer must not write to the configuration registers of an active DMA, because that will change the behavior of the active channel. It is okay to setup the registers for another channel which is not active.

3.1.3 Initialization

A few things must be done in the initialization routine:

1) Select the type of memory to use
2) Initialize that type of memory
3) Set up some most commonly used parameters so they don't have to be set everytime.

- Select the type of memory to use

They System Memory Interface supports various configurations of SRAM and SDRAM, with 8-bit or 16-bit external bus. These different types of memory is configured by the register *SDRAM_INIT*. This register must be set everytime PALM-I boots up to reflect the external system memory configuration. Once the memory type has been configured, the PALM-I chip behave the same way for all memory configurations, except the time required to read/write a data is different.

- Initial that type of memory.

SDRAM must be initialized, which is set at the same time as the memory type using *SDRAM_INIT*. In addition, the SDRAM refresh cycle (*SDRAM_REFH, SDRAM_REFL*) must be set. The refresh cycle is dependent on the clock frequency, so it might need to be set again when the clock frequency changes.

- Set up some most commonly used parameters so they don't have to be set everytime.

The common paramters include:
Width registers (*SDRAM_WID0L, SDRAM_WID0H,* etc)
Beginning and end of each FIFO (*CHN1_START_ADDR, CHN1_END_ADDR,* etc.)
CRITICAL, START, STOP values for each FIFO (*CHN1_HIGH_CHECK, CHN1_LOW_CHECK, CHN1_STOP_CHECK*).
Request levels for controller buffer, DSP PM, DSP DM. (*SDRAM_CON_REQ, SDRAM_DSP_REQ*)
Minimum transfer length (*SDRAM_MTL*)
Note: Some of these registers have defaults values that can already be used.

3.1.4 Setup DMA to/from Controller Buffer

Note: Only the 512B section from controller address 0x7000 to 0x71FF can be DMA'ed.

To set up a 1D controller DMA:

1) Initialization routine is finished..
2) Set whether the transfer is read or write and new request level (*SDRAM_CON_REQ*)
3) Make sure the memory interface has access to the controller buffer (*SRAM_CON_SWAP*).
4) Set the starting address of the DMA in the controller buffer (*SRAM_CON_CH1ADDRL, SRAM_CON_CH1ADDRH*). Set 0 to this register means DMA starts from address 0x7000, set 0x100 means DMA starts from address 0x7100, etc.
5) Set the length of the DMA transfer (*DMA9_X1*)
6) The Y value does not have to be set since the value is 0 from the previous DMA.
7) Set the *DMA9_HIGH* register depending on the high bits of the length of transfer and the high bit of the starting address. The width register is irrelevant since there is only one line DMA'ed.
8) Set the rest of starting address of the DMA in system memory. This is done by setting registers (*DMA9_START_ADDR3, DMA9_START_ADDR2, DMA9_START_ADDR1*). Remember to write *START_ADDR1* last since that will start the DMA.
9) Check the status of the DMA or wait for interrupt for completion.

For a 2D controller DMA, steps 5-7 should be:

5) Set the length of the DMA transfer (*DMA9_X1*).
6) Set the Y value of the DMA transfer*(DMA9_Y1)*.
7a) Set the width register if the desired value is not already programmed in.
7b) Set the *DMA9_HIGH* register depending on the high bits of the X value, Y value, and address, as well as to select the appropriate width register.

3.1.5 Setup DMA to /from DSP (PM and DMX)

Note: Only the sections from address 0x400-0x7FF can be DMA'ed for both DSP PM and DM.

To set up a 1D DSP DMA:

1) Initialization routine is finished, including setting the DSP buffers request level to non-zero.
2) Make sure the memory interface has access to the appropriate buffer. This is done by the Swap Memory Buffer command in the command queue.
3) Set the starting address of the DMA in the DSP buffer (*SRAM_DM_CH1ADDRL, SRAM_DM_CH1ADDRH, SRAM_PM_CH1ADDRL, SRAM_PM_CH1ADDRH*). Set 0 to this register means DMA starts from address 0x400, set 0x10 means DMA starts from address 0x410, etc.

4) Set the length of the DMA transfer (*DMA?_X1*)
5) The Y value does not have to be set since the value is 0 from the previous DMA.
6) Set the *DMA?_HIGH* register depending on the high bits of the length of transfer and the high bit of the starting address. The width register is irrelevant since there is only one line DMA'ed.
7) Set the rest of starting address of the DMA in system memory. This is done by setting registers (*DMA?_START_ADDR3, DMA?_START_ADDR2, DMA?_START_ADDR1*).
8) Start the DMA by issuing a command queue instruction (*CMDQ_DL, CMDQ_DH*). This will also set whether the DMA is for DM or PM, and whether the DMA is a read or write.
9) Check the status of the command queue or interrupt for completion.

For a 2D controller DMA, steps 4-6 should be:

4) Set the length of the DMA transfer (*DMA?_X1*).
5) Set the Y value of the DMA transfer *(DMA?_Y1)*.
6a) Set the width register if the desired value is not already programmed in.
6b) Set the *DMA_HIGH* register depending on the high bits of the X value, Y value, and address, as well as to select the appropriate width register.

Notes:
- The completion of a DMA to the DSP buffer will trigger a decrease in the number of commands in the command queue.
- The endian of the input values to DM and whether the 24-bit PM is left or right justified can be configured using the register *SRAM_DM_PM_CTRL*.

3.1.6 DMA to/from FIFO

External devices which require access to system memory data must go through a FIFO, which acts as a buffer since SDRAM data is very bursty. There are two pieces of physical memory, one for devices that write to system memory, and one for devices that read from system memory. This means that there can only be one active channel in each direction. The only exception is the stereo mode codec. In this mode, extra logic has been added so the left and right stereo channels can co-exist. In our application this limitation is okay since the different devices that use the DMA channels are never on at the same time. In fact, several devices share DMA channels because they can never be on at the same time.

To set up a DMA to a FIFO:

1) Initialization routine is finished.
2) Reconfigure the FIFO start/end, high/low/stop if necessary
3) Set the length of the DMA transfer (*DMA?_X1*)
4) The Y value does not have to be set since the value is 0 from the previous DMA.
5) Set the *DMA?_HIGH* register depending on the high bits of the length of transfer and the high bit of the starting address. The width register is irrelevant since there is only one line DMA'ed.
6) Set the rest of starting address of the DMA in system memory. This is done by setting registers (*DMA?_START_ADDR3, DMA?_START_ADDR2, DMA?_START_ADDR1*). Writing to *DMA?_START_ADDR1* will start the DMA.
7) Check the status register or wait for interrupt for completion.

For a 2D controller DMA, steps 3-5 should be:

3) Set the length of the DMA transfer (*DMA?_X1*).
4) Set the Y value of the DMA transfer *(DMA?_Y1)*.
5a) Set the width register if the desired value is not already programmed in.

5b) Set the *DMA_HIGH* register depending on the high bits of the X value, Y value, and address, as well as to select the appropriate width register.

A *FLUSH* register is added for FIFO's to system memory. This is for flushing out whatever is left in the FIFO when the request is zero.

3.1.7 BYTE access mode

All of the PALM-I accesses to system memory is based on DWORD boundaries. However, a special mode (called BYTE access mode) has been added to the system memory to allow reading starting from non-DWORD-aligned BYTE's. What this mode does essentially is read two DWORD, and piece together one DWORD that is not aligned. It will store the other BYTE's to be combined with the next DWORD. See the Chip Spec for more detail.

In order to set up a DMA using BYTE access, several factors must be determined:

1) Which channel will use the BYTE access mode. Only one channel can use this mode at a time. Because multiple active DMA's can be on at the same time, the channel that has the BYTE access mode must be specified.
2) Which byte to start from
3) Whether the first DWORD will be suppressed for each time.

These two modes are all set by one register: *SDRAM_BYTE_OP*.

3.1.8 Interrupts and Status

The status of each DMA channel can be read from *SDRAM_STATUS1* and *SDRAM_STATUS2*, which will show whether each DMA channel is active. Writing a 1 to a particular bit to these registers will forcefully disable that DMA.

A DMA can also generate an interrupt upon finishing by the following steps:

1) Enabling the interrupt from DMA in the register *INT_EN*, bit 1.
2) Enable the channels that should interrupt the controller when their DMA is finished by configuring registers *SDRAM_DMA_EN1* and *SDRAM_DMA_EN2*.
3) After the interrupt occurs, the channel that caused the interrupt (if there are multiple possible sources) can be read from *SDRAM_DMA_INT1* and *SDRAM_DMA_INT2*.
4) Clear the interrupt by writing a 1 to that channel's bit location in *SDRAM_DMA_INT1* and *SDRAM_DMA_INT2*.

3.1.9 DSP Direct Access to DMA

To improve the efficiency for a DMA to DSP buffers, PALM-I allows the DSP to setup a DMA directly. This is done by mapping the registers for the DSP DMA channels to a portion of DSP DMX (i.e. starting from DMX 0x200). To enter this mode, the controller must write a 1 to the *DSP_MODE* bit in the *CMDQ_PCH* register. After this is done, only DSP execution commands can be issued by the controller. All the other commands such as load/store/swap is handled by the DSP directly.

Once the DSP enters this mode, it sets up a DMA by setting up registers such as starting address, X, Y, width, in a similar fashion as the controller. The paramters that the DSP can read/write include:

- Starting address
- X, Y
- Starting address in PM/DMX
- Width and width register select
- Interrupt enable, status, and clear

- Channel status
- DSP DMA request level
- DSP DMA Byte access
- Endian, PM left/right justified, single/double buffer

The DMA is started by writing to the *DSP_COMMAND_REGISTER*. This is similar to writing to the command queue by the controller, except only one command can be written at a time, and the only commands which are valid are the load/store commands and the swap command. Once the command is completed, the top bit of the *DSP_COMMAND_REGISTER* is cleared. The DSP can use that as a status bit to check whether to continue to the next command.

3.2 Sensor Interface Programming

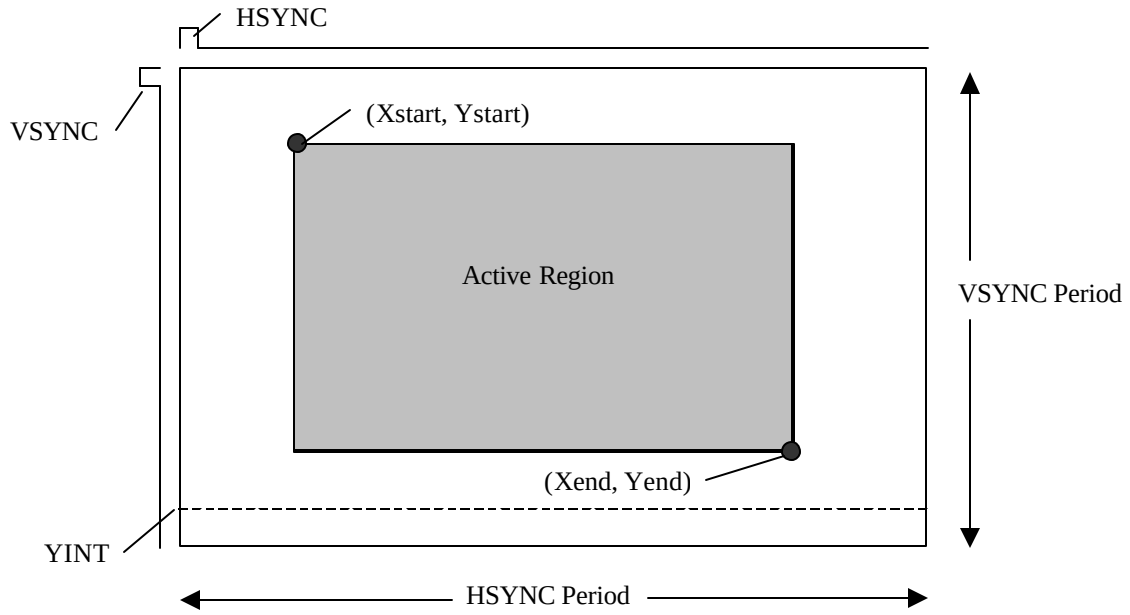
3.2.1 Control Signals

The sensor interface is a slave-mode only interface with three main control signals (all inputs): PIXCLK, HSYNC, VSYNC. PIXCLK toggles once for every pixel. HSYNC toggles once at the beginning of each line, and VSYNC toggles once at the end of each line. These signals define the size of one "frame" of picture. The actual active region is defined with respect to this frame (see below). To allow the most flexibility in the type of sensor to use, the programmer can set the polarity of these three signals, i.e. to make HSYNC/VSYNC active HIGH/LOW and to latch pixel values on the rising or falling edge of PIXCLK.

3.2.2 Defining an active region

The sensor will output a 2-D image, with HSYNC defining the beginning of a new line, and VSYNC defining the beginning of the image. The sensor interface will keep track of the number of pixel clocks after each HSYNC (X count) and the number of lines after VSYNC (Y count). (The current X count and Y count can be read from registers *CAM_XCOUNTL, CAM_XCOUNTH, CAM_YCOUNTL, CAM_YCOUNTH*). The most important for the programmer to do is to define active region, in which the image is valid and the pixel data is store to system memory via a FIFO. The pixels outside of the active region is ignored. The active region is defined by defining an X start and X end, and a Y start and Y end. In all the lines between Y start and Y end, if the pixel is between X start and Xend, then the pixel is valid. These values are defined using the registers *CAM_STARTL, CAM_XTARTH, CAM_XENDL, CAM_XENDH, CAM_YSTARTL, CAM_YSTARTH, CAM_YENDL, CAM_YENDH*. Because the sensor interface is slave-mode only, the sync signals are generated by the CMOS sensor, so the HSYNC period and VSYNC period are determined externally.

Another thing the user can set is a interrupt line. The sensor interface will then generate an interrupt whenever it reaches that line. Usually this image is defined for before or after the active region, and can be used to as a frame interrupt for video applications or do some setup before or after an image capture. The interrupt line is set using the registers *CAM_YINTL, CAM_YINTH*. Of course, the sensor interrupt must be enabled in the *INT_ENABLE* register, and the interrupt must be edge-triggered (*INT_EDGE*).



3.2.3 Camera I/O and I2C interface

There are four programmable I/O that is dedicated to the sensor interface. They are controlled by the registers *CAM_IO_EN* and *CAM_IO_DATA*. These can implement functions such as SENSOR SLEEP/RESET or implement a simple serial protocol such as I2C to read/write data from/to the sensor. The top two bits of the IO are default outputs with default values of 10 and the bottom two are inputs.

One of the important functions of the I/O is to keep the sensor in reset or sleep state while the sensor interface is being set up. The programmer must have the polarity of the SYNC signals, the active region all defined before the sleep/reset can be disabled and the sensor will start sending out information. Otherwise, the captured image could be anything.

3.2.4 FIFO

The sensor interface pads each 10-bit pixel data with 6 leading zeros and packs two pixel data into one DWORD to be sent to the FIFO and then to System Memory. The sensor uses DMA channel 3. Refer to the DMA section to see how to set up the FIFO and the DMA. Note that the sensor FIFO needs to be flushed at the end to insure the last few pixels of the image is written out.

3.3 Codec Interface Programming

3.3.1 Codec Types

The PALM-I Chip supports two types of codecs: Philips UDA codec and AC97 codec. These two have very different protocols and thus are implemented as separate interfaces. However, they share I/O pins and FIFO channels, so they cannot both be used at the same time. During initialization, the type of codec is determined by the register *CODEC_SHARE*. Also the bits for USBout_Sel and USBin_Sel must be disabled for the FIFO's to be used by the codec.

3.3.2 UDA

The UDA codec outputs both the BITCLK and WS signals. These signals must be programmed with the register *UDA_CLOCK_SET*, which sets both the number of system clock cycles per BITCLK cycle and the number of BITCLK cycles per WS.

The UDA also supports two data modes – I2S and 16-bit LSB-justified mode. See the codec interface for more details. This mode is set using the *UDA_MODE_SET* register. The *UDA_MODE_SET* register also sets whether the input/output channels are valid. When DA channels are valid, then the codec reads a 32-bit DWORD from the FIFO as one left sample and one right sample. One only one channel is valid, then the 32-bit DWORD represents two data on the same channel. When both channels are disabled, then no data is read. The same is equivalent for AD channel valid.

The UDA codec supports an additional L3 interface for writing control registers in the codec. This is implemented with a 3-bit I/O (*UDA_L3CLK, UDA_L3DATA*). The exact protocol can be found in the codec spec.

3.3.3 AC97

The AC97 interface has BITCLK as input and outputs a SYNC signal (which shares a pin with the WS signal of the UDA codec). The AC97 supports a warm wake, which is implemented by writing a one to *AC97_WARMWAKE* and then clearing that register after a short while. The AC97 interface is a multiple time-slot interface, which includes both command address, data and DA/AD data. Whether each time-slow is valid depends on a VALID bit in the first time-slot.

To write a register in the AC97 interface, the programmer needs to set the command address register (*AC97_CMDADDRL, AC97_CMDADDRH*) and the command data register (*AC97_CMDDATAL, AC97_CMDDATAH*). And then the command can be sent by setting the VALID bits for the entire frame and the command address and data in register *AC97_DASLOT*.

To read a register in the AC97 interface, the corresponding command and address must be sent to the codec, and the codec will respond with the data in the A/D line. The data can be read in register *AC97_STATDATAL* and *AC97_STATDATAH*. The program must wait for long enough for the read command to be sent to the AC97 and the value be sent back.

Two of the valid bits in *AC97_DASLOT* are for the left and right D/A channels. When these bits are set, then the codec reads the data in those time-slots as real data. However, another register controls whether each channel actually reads data from the FIFO. That register is *AC97_DAVAL*. When both channels are valid, then the 32-bit DWORD represent one left sample and one right sample. When only one channel is valid, then each DWORD represents represent two data samples on the same channel. When both channels are disabled, then no data is read.

The equivalent is true for A/D data and registers *AC97_DAVAL*. Of course, *AC97_ADSLOT* is read-only since the codec decides which data is valid.

In the AC97 mode, the L3 pins for the UDA codec can be used as general purpose I/O.

3.3.4 DMA Channels

The D/A output of the codec can be a stereo output, with the left and right channel data in separate locations. This prompted the stereo mode of the codec FIFO, which allows two DMA's (channel 4 and channel 6), one from the location of each channel. These DMA's go in parallel, and the data can be pieced together and sent to the codec as one 32-bit value.

The stereo mode is set by setting bit 0 of *CODEC_SHARE*. After the stereo mode bit is set, the FIFO will read one DWORD from each FIFO and piece them together for two DWORD data. The combination of the data depends on

the values of bits 3 and 2 of *CODEC_SHARE*. The following is a list of the different combinations, assuming that the two channels contain AB and CD respectively.

Bits 3:2	First Data to Codec	Second Data to Codec
00	AC	BD
01	CA	DB
10	BD	AC
11	DB	CA

Basically, these two bits selects which channel is left and which is right and the endian of the data.

Of course, two separate DMA's must be set for channels 4 and 6 for the respective audio data. There is only one physical piece of memory for the two FIFO's, so when defining the start and end for these two FIFO's care must be taken so they do not overlap. After the two FIFO's are defined, then the two DMA's are started like normal DMA's.

If the stereo mode is disabled, then the data is taken directly from DMA channel 6, and bit 2 of *CODEC_SHARE* can still select whether the top WORD or bottom WORD is used by the codec first.