

---

*The FPAA interface: Software to make things automagical.*

## 1 Overview

The 2.8x RASP FPAA boards uses an AT91SAM7S to communicate via USB to the any desktop computer. The software emulates a serial CDC connection, and most modern operating systems have drivers for this software out-of-the-box. This document describes software version 0.0.6 on FPAA board revision 2. This document was last updated October 30, 2008.

## 2 Startup for the impatient.

Plug the device into the computer. If you are in windows, the device will show up as a COM port, and then you can use HyperTerminal to communicate to the device to see if it is responsive. The BAUD settings are arbitrary. If you are in MacOS or \*nix, you will see the device mount as /dev/tty.usb\_something in the device tree. You can then send commands to see if the device is operating properly. A good starting command is:

```
setdac(0,0x3fff);
```

This command will set the DAC on channel 0 to output a voltage of 5.0 volts.

## 3 Commands

The software responds to text commands and always returns a 32-bit value. All commands are in the format of “command(arguments);”. The arguments may be integer or float values dependent of the function. The software automatically assumes that arguments which start with “0x” are hexadecimal numbers, otherwise decimal numbers are assumed.

If you wanted to set the onboard 14-bit DAC of be 4.1 volts, you would send the command “setdac” and see the response in a terminal. “setdac” takes the channel and then the voltage as arguments.

```
setdac(0,4.1);  
0x000030a3
```

The returned value is calculated value for 4.1 volts on the 5 volt DAC. If you wanted to set a specific voltage without the risk of the internal FPU emulation, you would send the following:

```
setdac(0,0x30a3);  
0x000030a3
```

### 3.1 Command format.

The command format looks much like what one would expect from a function in C, with the generic definition of “int function(arg1, arg2, ...);”. Unlike C, the arguments can be mixed due to the nature of this being a command interpreter. Hexadecimal values start with “0x” and integer values are

denoted from decimal values by the decimal. The functions always return a 32-bit hexadecimal string in the form of “0x12345678”. If a command is not found, a -1 as “0xffffffff” is returned. If a command has more or less arguments than expected, a -2 as “0xffffffe” is returned. *note:* The software does not handle queued commands; therefore, one should be sure that data was returned before writing again to the device.

### 3.2 FPAA Command List

The following functions control the FPAA internal SPI registers and values.

#### fg\_select()

The *fg\_select* function controls the selection of floating-gates in the FPAA. The function is overloaded, and takes either a row address and a series of switches or a hex value. If one argument is detected, the function expects a 156-bit hexadecimal, otherwise three or more arguments are expected from which the 156-bit control word is generated.

The *fg\_select* takes arguments as row, (col, col, col) in a comma separated series, and an I-V select bit: “fg\_select(row,(col, col), I-V);” The row value is a 9-bit value for the row index. The valid value for columns are from 143-0. For example, if one wanted to select row 2, the columns 0,143,23,82, and use the fine-grained I-V circuit, one would pass: “fg\_select(2,0,143,23,82,1);”. The column select order is not important as the same mask is generated regardless of the column address order.

lexer.c parses and controls clearing the masks, but commandcontrol.c takes care of bitmask assembly. An excerpt from lexer.c follows.

```
l_fgs_clearcols(); //zero the column masks
a_1 = command_getarg(buffer,bufferlen,1); //get the row
if (ishexstr(c_stack,a_1)==1) //convert HEX text to int if necessary
{ a_1=hexstrtohex(c_stack,a_1);
}else{ a_1=strtohex(c_stack,a_1);}

a_2 = command_getarg(buffer,bufferlen,i_tmp); //get the IV sel flag
if (ishexstr(c_stack,a_2)==1)
{ a_2=hexstrtohex(c_stack,a_2);
}else{ a_2=strtohex(c_stack,a_2);}

//now we have to loop to generate out bitmaks for the columns
for(i=2; i< i_tmp;i ++)
{
a_3 = command_getarg(buffer,bufferlen,i);
if (ishexstr(c_stack,a_3)==1)
{ a_3=hexstrtohex(c_stack,a_3);
}else{ a_3=strtohex(c_stack,a_3);}
l_fgs_select_column(a_3);
}

//now that we are done, call the latch function
i = fpaa_matrix_latch(a_1,a_2);
```

The *fg\_select* takes a hexadecimal string of 156-bits as an argument. The last 2 bits of the of the 156-bits is ignored because the FPAA requires 154-bits. The bits are clocked in from left-to-right. For instance, if the argument started with 5, we’d have: “fg\_select(0x5XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX);”, where the first data seen on the SPI line would be the binary “0101” of the 0x5.

For implementation details, see the arbitraryspibits() function in controlcommands.c.

**fg\_gdac()**

The *fg\_gdac* command is an overloaded function that controls the internal gate DAC of the FPAA. It takes a hexadecimal value which translates into 7-bits for the FPAA gate dac, or a hexadecimal value DAC value and whether to broadcast the value externally. For example, the command “fg\_gdac(0x7f);” would rail the gate DAC high. The FPAA gate DAC needs to be trimmed via channel 39 on the off-chip DAC. An example of this is “setdac(39,0x028f);”. The *fg\_gdac* does not automatically trim the DAC because the value is arbitrary. “fg\_gdac(0x7f);” is equivalent to “fg\_gdac(0x7f,0);”.

For implementation details, see the `fpaa_gatedac()` function in `controlcommands.c`.

**fg\_ddac()**

The *fg\_ddac* function controls the internal drain DAC of the FPAA. It takes a hexadecimal value which translates into 7-bits for the FPAA drain dac, or a hexadecimal value DAC value and whether to broadcast the value externally. For example, the command “fg\_ddac(0x7f,1);” would rail the drain DAC high and broadcast the output to the drain pin.

For implementation details, see the `fpaa_draindac()` function in `controlcommands.c`.

**fg\_inj()** The *fg\_inj* function controls the programming of a switch the FPAA. It takes three arguments: the 7-bit gate voltage as either decimal or hexadecimal, the 7-bit drain voltage as either decimal or hexadecimal, and the time to pulse the drain. For example, “fg\_inj(0x70,0,200);” will put a gate value near  $V_{dd}$  on the gate, GND on the drain, and pulse the injection for  $100\mu S$  (see the pulse IO function for time mappings). This function does not support voltage conversions in decimal form, which makes this different from the *setdac()* function because the resistors were still a bit of an unknown at the time of this writing.

For implementation details, see `controlcommands.c`.

**fg\_ramp()** The *fg\_ramp* function does a controlled stepping of voltage from a starting voltage to an ending voltage for a given number of steps. This command takes either three or four arguments: starting voltage or 14-bit hexadecimal, ending voltage or 14-bit hexadecimal, the number of steps, and optionally, which DAC channel to use. The default  $A_{vdd}$  has a gain of 2, so this is divided internally by one half to make 2.4 actually 2.4 when the default channel is used. For example, “fg\_ramp(2.4,3.3,100);” will cause the  $A_{vdd}$  to travel from 2.4v to 3.3v at a granularity of approximately 100 steps. The function rounds based on the lowest value support by the DAC. This function is hard-coded to use channel 6 by default if only three arguments are given, as that was mapped to  $A_{vdd}$  on the equipment available. Another example, “fg\_ramp(2.4,5.0,100,1);” will cause channel 1 to travel from 4.8v to 10.0v at a granularity of approximately 100 steps. Using the fourth argument is inherently dangerous and should be avoided.

For implementation details, see `controlcommands.c`.

**fg\_adc()**

The *fg\_adc* samples the ramp ADC on the FPAA. It is overloaded and takes no arguments or one argument which specifies the maximum allowed time to ramp. If no value is given, this is functionally equivalent to “fg\_adc(0);” which does not constrain the number of samples, but has a

ability to freeze the MCU if there is not an appropriate bias set on DAC channel zero. The function works by setting pin 29 high and waiting for charge to integrate on a capacitor until line 26 goes high. The function returns an arbitrary amount clock cycles which can be used create a lookup table for different voltages. One should also note that the IV bit in the *fg\_select* register should be set to 1. This done via “fg\_select(0,33,1);” internally in the function as a floating gate does not exist at that address.

For implementation details, see the `fpaadraindac()` function in `controlcommands.c`.

### 3.3 Board Command List

The following commands control the devices on the FPAA board (section 4), such as the 40-channel DAC (section 4.2) and 8-channel ADC.

**setdac()** The *setdac* command controls the 14-bit, on-board AD5380 DAC and takes two arguments. The command expects arguments of channel and line value: `setdac(<0-39>,<0-0x7fff>)`; The command will take decimal values for the second argument and attempt to convert the decimal to a valid hexadecimal input for the DAC, for example “setdac(1,5.0);” should set channel 1 to be 0x3fff for a unity gain channel. Table 4.2 lists DAC channels and gain on each channel.

For implementation details, see the `setdac()` function in `controlcommands.c`.

**readadc()** The *readadc* command reads an 8-bit value from an ADC channel. The function takes one argument that is the ADC channel.

For implementation details, see the `readadc()` function in `controlcommands.c`.

**setout()** The *setout* command takes one argument, and sets one of the AT91SAM7 lines (section 4.1) and an output. For example, “setout(19);” will set PA19 to be an output.

For implementation details, see the `setout()` function in `controlcommands.c`.

#### **setin()**

The *setin* command takes one argument, and sets one of the AT91SAM7 lines (section 4.1) and an input. For example, “setin(19);” will set PA19 to be an input.

For implementation details, see the `setin()` function in `controlcommands.c`.

#### **readio()**

The *readio* command takes one argument, and retrieves the binary one of the AT91SAM7 I/O lines (section 4.1). The command returns a zero if the line is not set, and a non-zero value when set. For example, “readio(19);” will return a 0x00080000 representing the 19th bit if the input has a logical “1”.

For implementation details, see the `readio()` function in `controlcommands.c`.

#### **setio()**

The *setio* command takes two arguments, the pin and the logic level. “setio(19,0);” will set PA19 to a logic level of zero.

**isin()**

The *isin* command takes one argument, and returns a non-zero value if the pin is set to be an input (section 4.1). For example, "isin(20);" will return a 0x00100000 representing the 19th bit if the pin is configured to be an input.

For implementation details, see the *isin()* function in *controlcommands.c*.

**toggle()** The *toggle* command takes one argument, and will toggle the specified I/O line (section 4.1). This command combined with *spisel* can be used drive the SPI clock for debugging.

For implementation details, see the *toggle()* function in *controlcommands.c*.

**pulseio()** The *pulseio* command takes two arguments, the pin number and the duration. This command will toggle the specified I/O line (section 4.1) for the duration. This command is at the core of the *fg\_inj()* function. The function returns the base seed for the counter. A mapping for the duration argument compared to real-time is in the following table.

duration	time (seconds)	magnitude
0	2	-6
10	7	-6
200	100	-6
2000	1	-3
10000	5	-3
1000000	500	-3

This function is inherently unsafe because you have the ability to loop long enough to stall the USB bus. At  $500mS$ , one should understand that they are living dangerously, so if long durations are required, the pulse should be split into a series of separate pulses. *note:* The theoretical minimum for a pulse is  $600nS$ ; however, if this is desired a new function must be written. The stack overhead causes this function to currently have minimum duration of  $2\mu S$ , but it makes a nice even divider for pulse timings. I did instruction counting of the stack frames to achieve this timing.

For implementation details, see the *toggle()* function in *controlcommands.c*.

### 3.4 Debug Command List

**output()**

The *output* command takes one argument which will enable or disable the return of data. The function format requires that every command return data, so *output* allows one to suppress the output of commands such as *setdac* by using "output(0);"; however, this will also suppress outputs from commands such as *readadc*. "output(1);" should be set before attempting to read information back from the controller.

For implementation details, see the *util\_setoutput()* function in *util.c*.

**version()**

The *version* command takes no arguments and returns the software version as a 32-bit representation, 0xAABBCCDD, which represents a version in BB.CC.DD. AA should be 0. BB is the major version number. CC is the minor version number. DD is the revision number. If the command returns 0x00010301, the value would represent version 1.3.1.

For implementation details, see the `software_version()` function in `controlcommands.c`.

**spisel()** The *spisel* command selects one device via decoder on the SPI line. The argument should be either in hex or a decimal.

device description	SPI address	c #define
FPAA gate DAC	0x00	SET_GATEDAC
FPAA drain DAC	0x01	SET_DRAINDAC
FPAA bits register	0x03	SET_FPAA
Board DAC AD5380	0x04	SET_DAC
Board ADC AD7808	0x05	SET_ADC

For implementation details, see the `spibang_sel()` function in `spi.c`. The c defines are in `controlcommands.h`

**spilatch()** The *spilatch* command sets the latch line on the SPI bus. The latch line is active low, so one should set the latch line high when not in use. The FPAA has a broken SPI bus so you go low after all of the bits are clocked; whereas, every other device you assert the latch line for the duration of the time that SPI bus is active. The FPAA also latches on the rising edge, so the clock must be set high before SPI transfers to the FPAA start.

**spiwrt()** The *spiwrt* command allows one to easily put arbitrary bits on the SPI bus. The command takes a data, a mask and whether the clock should start high or low. For example, `"spiwrt(0x0000f0f1,0x00001003,1);"` would write only 3 bits to the bus because only 3 bits are masked. The bits are placed on the bus MSB to LSB in network byte order notation where the MSB is on the left in the example above. The function can place a maximum of 32 bits on the bus due to fact that it is a direct interface into SPI calls.

For implementation details, see the `spibang_sel()` function in `spi.c`.

**testarg()** The *testarg* commands takes a hexadecimal or decimal value as an argument and returns what the string parser would pass to a command.

For implementation details, see the `hexstrtohex()` and `strtohex()` functions in `lexer.c`.

**d()** The *d()* takes no arguments and is a hook into the `x_debug()` function in `controlcommands.c`. The function exists for convenience as `lexer.c` is inherently messy. One should never run this unless they really, really know what they are doing.

**m()** The *m()* takes one argument and is used to return mask information to the readable world.

### 3.4.1 Examples of I/O with debug commands

As an example of how to use the debug commands, let one assume that the FPAA drain DAC is to be used to broadcast a voltage. First, the basic commands to bring up the board. In the following, C-style comments are used, even though they are not supported by the software.

```
setout(21); //PULSE the line which is pulsed high to apply a drain pulse to a
           floating gate
setout(24); //ON_CHIP global logic which tells if you are using on-chip stuff.
setout(27); //PROG program mode or run mode. prog mode is active high
setout(29); //MEASURE turn it high to connect the drain of the selected device to the I-V.

setio(21,0);
setio(24,1);
setio(27,0); // The PROG line actually is what drives the DAC to an external pin,
           start low for the example.
setio(29,0);

setdac(6,1.2); //set AVDD, to a power amp with a gain of 2, so 1.2 puts 2.4v on the FPAA
setdac(39,0.22); //dactrim. This value varies with IC, but 0.22v seems to be about correct.
setdac(9,1.8); //vref for the DACs

//now on to SPI
setsel(0); //the spi address of the gate dac
spiwrt(100,0x0000007f,1); //send a decimal value of 100 across 7 bits (7f), and
start with the clock line high (1) because the FPAA is rising edge latching.
spilatch(0);
spilatch(1); //latch the data

setio(27,1); //broadcast the gate dac voltage off-chip
```

### 3.5 Matlab Example

The following is an example for setting DAC voltages via MATLAB under Windows. Richie contributed the following function.

```
function dac(varargin)
%dac(varargin)
%   varargin =
%           'init' - initializes the dac
%           'stop' - closes the connection to the dac
%           channel_x, value_x, channel_y, value_y, ... - sets the
%                   voltage value_x onto channel_x etc
%ex.
%dac('init')
%dac(1,1.123, 7,2.314, 20,4, 21,4, 15,0)   %sets:  dac1 = 1.123
%                                           dac7 = 2.314
```

```
%                               dac20 = 4
%                               dac21 = 4
%                               dac15 = 0

persistent sc;
if strcmp(varargin{1}, 'init')
    sc = serial('COM5', 'BAUD', 9600);
    fopen(sc);
    return
end
if strcmp(varargin{1}, 'stop')
    fclose(sc);
    return
end

c = (1.6128*10^4)/4.9207;    %dec to hex plus calibration junk

if mod(nargin,2) == 1
    error('arguments should be in pairs');
end

for i = 1:2:nargin
    channel = varargin{i};
    val     = varargin{i+1};

    setStr = sprintf('setdac(%g,0x%s)', channel, dec2hex(round(val*c)));
    %fprintf('%s\n', setStr);
    fprintf(sc, '%s\n', setStr);
end
```

## 4 Board Description

### 4.1 MCU pin mappings

SAMDIP-7S GPIO mappings				
I/O Pin	Intent	Default	Purpose	Description
PA0	Output	0	SPI select	Address decoder multiplexor
PA1	Output	0	SPI select	
PA2	Output	0	SPI select	
PA3	Output	1	Tunneling	Active low line to override global tunneling circuitry
PA11	Output	1	SPI control <sup>1</sup>	latch line for valid SPI data
PA12	Input	-	SPI data	peripheral data to the mcu
PA13	Output	0	SPI data	data from the mcu
PA14	Output	0	SPI clock <sup>2</sup>	clock for SPI data
PA15	Output	0	USB	
PA16	Input	-	USB	
PA17	Output	1	FPAA Select	?
PA18	Output	1	FPAA Drain	?
PA19	Output	1	FPAA Gate	?
PA20	Output	1	FPAA Reset	Active low reset for FPAA components.
PA21	Output	0	FPAA Pulse	Raise high to pulse the drain of a floating gate.
PA23	Output	1	FPAA ADC	ADC clock ?
PA24	Output	1	FPAA On-chip	Select whether on-chip peripherals are used.
PA26	Input	-	FPAA Ramp	Ramp ADC input, high when conversion complete.
PA27	Output	0	FPAA Prog	Low is run mode. High is program mode.
PA29	Output	0	FPAA Measure	High to connect device drain to I-V circuit.

<sup>1</sup>The FPAA uses a special scheme to latch the SPI data. As it does not have an internal counter, it accepts all data and the latch line must be asserted low and then high for data to be latched. This is different from the other SPI devices where the line must be asserted low for the whole data transfer and is latched on the rising edge.

<sup>2</sup>There is quirk with FPAA clock were the clock must start HIGH before the first bits are clocked because the FPAA latches on the rising edge of the clock. The other devices on the SPI latch on the falling edge.

## 4.2 DAC Channels

DAC mappings				
Channel	Name	Gain	Default	Purpose
0	–	1	0.0v	User specified
1	–	1	0.0v	User specified
2	DRAIN	2	0.0v	??
3	GATE	2	0.0v	??
4	TUNNELCAB	3	0.0v	??
5	TUNNEL	3	0.0v	??
6	AVDD	2	1.2v	Sets analog $V_{dd}$ to 2.4v.
7	DACTRIM	2	0.0v	??
8	–	1	0.0v	User specified
9	–	1	0.0v	User specified
10	–	1	0.0v	User specified
11	–	1	0.0v	User specified
12	–	1	0.0v	User specified
13	–	1	0.0v	User specified
14	–	1	0.0v	User specified
15	–	1	0.0v	User specified
16	–	1	0.0v	User specified
17	–	1	0.0v	User specified
18	–	1	0.0v	User specified
19	–	1	0.0v	User specified
20	–	1	0.0v	User specified
21	–	1	0.0v	User specified
22	–	1	0.0v	User specified
23	–	1	0.0v	User specified
24	–	1	0.0v	User specified
25	–	1	0.0v	User specified
26	–	1	0.0v	User specified
27	–	1	0.0v	User specified
28	–	1	0.0v	User specified
29	–	1	0.0v	User specified
30	–	1	0.0v	User specified
31	–	1	0.0v	User specified
32	–	1	0.0v	User specified
33	–	1	0.0v	User specified
34	–	1	0.0v	User specified
35	–	1	0.0v	User specified
36	I2VTRIM	2	3.3v	trim voltage for the I-V log amp
37	ADTRIM	1	3.03v	trim voltage for FPAA ramp ADC
38	ICHAR	1	2.7v	input into ramp ADC for characterization
39	DACVREF	2	1.8v	$V_{ref}$ for the FPAA DAC

## 5 Explanation of Sources.

The source code is heavily based upon the ATMEL example sources. The project compiles with the ARM tool chain through a Makefile. All of the example code has been heavily modified to support GCC<sup>3</sup> over the IAR code. Care was taken to make minimal changes to all code outside of what was specific to the FPAA. By typing “make” in the root directory, the code will automatically create a file called “main.bin” in the ./fpaa directory. The file can then be load to a clean AT91 via SAM-BA on Windows, or through the \*nix arm tools. sam7utils<sup>4</sup> were used under MacOS 10.3 during the course of development. The command on MacOS to flash the ARM is:

```
sam7 --exec set_clock --exec unlock_regions --exec "flash fpaa/main.bin"
```

### 5.1 Source Tree

The source tree as it looks as of 26 JUN 2008 follows.

```
-rwxr-xr-x  1 degs  degs    765  8 May 17:29 AT91SAM7S256_memory.ldh*
-rwxr-xr-x  1 degs  degs   4728  8 May 17:29 AT91SAM7S_sections_ROM.ldh*
-rwxr-xr-x  1 degs  degs  16233  9 Jun 13:30 Makefile*
drwxr-xr-x  6 degs  degs    204 26 Jun 20:35 cdc/
drwxr-xr-x 14 degs  degs    476 26 Jun 20:35 core/
-rw-r--r--  1 degs  degs     90  8 May 17:29 flash.sh
drwxr-xr-x 12 degs  degs    408 26 Jun 20:35 fpaa/
drwxr-xr-x  4 degs  degs    136  8 May 17:29 include/
-rwxr-xr-x  1 degs  degs 146538  8 May 17:29 lib_AT91SAM7S256.h*
drwxr-xr-x  8 degs  degs    272  8 May 17:29 linker/
drwxr-xr-x 16 degs  degs    544  8 May 17:29 msd/
drwxr-xr-x 10 degs  degs    340  9 Jun 13:24 old/
-rwxr-xr-x  1 degs  degs   12347  8 May 17:29 startup.S*
-rwxr-xr-x  1 degs  degs    2622  8 May 17:29 syscalls_gnu.c*
```

The “cdc” and “core” directories contain files which are very similar to the ATMEL example code with changes specific to GCC. All of the files which are specific to this project are in the “fpaa” directory. The “fpaa” directory contains the follow as of 26 JUN 2008:

```
-rw-r--r--  1 degs  degs   4760 23 Jun 13:42 controlcommands.c
-rw-r--r--  1 degs  degs    755 23 Jun 10:44 controlcommands.h
-rw-r--r--  1 degs  degs  20501 23 Jun 10:48 lexer.c
-rw-r--r--  1 degs  degs    804 21 Jun 17:29 lexer.h
-rwxr-xr-x  1 degs  degs  14537 23 Jun 09:48 main.c*
-rw-r--r--  1 degs  degs   2644 10 Jun 13:36 main.h
-rw-r--r--  1 degs  degs   4650 20 Jun 11:04 spi.c
```

<sup>3</sup><http://www.gnuarm.com/>

<sup>4</sup>available from <http://oss.tekno.us/sam7utils/download.php> at the time of writing. A search for “sam7utils” will show package locations for various OSes.

```
-rw-r--r--  1 degs  degs  1168 23 Jun 15:35 spi.h
-rw-r--r--  1 degs  degs   817 12 Jun 15:58 util.c
-rw-r--r--  1 degs  degs   192 11 Jun 11:15 util.h
```

### **main.c**

The “main.c” file is a modified version of the stock “serial\_example.c” from the “cdc” directory. The data stream is hijacked in the function which forwards USB to serial. The main() function also has all of the SPI setup calls. The function puts text characters into a stack which is then sent to the parser, lex.c, upon when a newline character occurs.

### **spi.c**

The “spic.c” is a bitbang implementation of SPI. The on-board SPI of the AT91SAM7 can only send 8-bit series words, and the FPAA requires odd bits. Furthermore, the FPAA does not follow SPI line protocol for latching. All of the SPI related functions reside in SPI.c.

### **util.c**

The “util.c” contains all of the utility functions and any assembly language. The file is compiled in assembly mode.

### **lexer.c**

The “lexer.c” file is the parser the text commands. The lex engine expects text followed by “(” and “)”. The parenthesis can hold an arbitrary number of arguments. The parsed command then is passed to “controlcommands.c” for command execution.

### **controlcommands.c**

The “controlcommands.c” file contains all actual commands. These commands call SPI and other support infrastructure.