

# **Programmer's Manual**

## **Digital Gamma Finder (DGF)**

### **POLARIS**

Version 3.30, August 2007

**XIA LLC**

31057 Genstar Road  
Hayward, CA 94544 USA

Phone: (510) 401-5760; Fax: (510) 401-5761  
<http://www.xia.com>



#### **Disclaimer**

Information furnished by XIA is believed to be accurate and reliable. However, XIA assumes no responsibility for its use, or for any infringement of patents, or other rights of third parties, which may result from its use. No license is granted by implication or otherwise under the patent rights of XIA. XIA reserves the right to change the DGF product, its documentation, and the supporting software without prior notice.

# Table of Contents

1.	POLARIS API Overview .....	1
2.	Polaris API Function Description .....	1
2.1.	C_Hand_Down_File_Names .....	2
2.2.	C_Boot_System .....	3
2.3.	C_User_Par_IO .....	5
2.4.	C_Acquire_Data .....	7
2.5.	C_Set_Current_Module .....	10
2.6.	C_Buffer_IO .....	11
	Options for Compiling POLARIS API.....	12

## 1. POLARIS API Overview

The Polaris API consists of a small set of functions aimed at performing a specific set of tasks.

At first the Polaris needs to be initialized. This is a process in which the Polaris is made known to the host computer by establishing communication via the EZ-USB interface. The function **C\_Boot\_System** is used to achieve this. Before calling this function, another function, **C\_Hand\_Down\_File\_Names**, should be called to let the API know the name of system configuration files (FPGA, DSP, and settings files). The boot process involves downloading the FPGA configurations, DSP executable code and system configuration parameters to the Polaris.

Now, the instrument is ready for data acquisition. Normal operation is to start the first MCA histogram data acquisition run using the NEW\_RUN mode to erase any old histograms and statistics information (**C\_Acquire\_Data**). Since this type of run does not time out by itself, the user must end it using **C\_Acquire\_Data** with the stop\_run run type. After the run is ended, histograms and statistics data are available in the Polaris memory for transferring to the host computer.

A secondary mode of operation is provided for diagnostic and calibration purposes. The system can be programmed to acquire untriggered ADC traces. The function to achieve this is again **C\_Acquire\_Data**. A user can check the ADC trace for DC-offset or signal gain.

After a successful MCA run, it may be necessary to adjust some parameters for optimal energy resolution or throughput. The function **C\_User\_Par\_IO** can be used to change a variety of system parameters such as filter times, preamplifier decay time, system gain or offset, etc.

Another diagnostic function provided to Polaris users is the function **C\_Buffer\_IO**, which can be used to read out the DSP internal memory, or writing DSP I/O parameters to the internal memory. The last function, **C\_Set\_Current\_Module** is only useful for a multi-Polaris system where the host can call this function to indicate which Polaris module it wants to communicate to.

## 2. Polaris API Function Description

The syntax, detail description and usage example of each Polaris API function are given in the following pages.

## 2.1. C\_Hand\_Down\_File\_Names

---

### Syntax

```
long C_Hand_Down_File_Names (  
    char *File_Names[]);           // A two dimensional string array  
                                   // containing the names of system  
                                   // configuration files
```

### Description

Use this function to download the names of files which are required for booting the Polaris. The API needs these file names so that it can read the Polaris hardware configurations from the files stored in the host computer and download these configurations to the Polaris.

This function must be called as the first step in the boot process.

### Return Values

Value	Description	Error Handling
0	Success	None

### Usage Example

```
char *boot_file_names[6] = {  
    "C:\\XIA\\Polaris\\Firmware\\sysg200.bin",  
    "C:\\XIA\\Polaris\\Firmware\\fg2004.bin",  
    "C:\\XIA\\Polaris\\dsp\\polaris_rc.bin",  
    "C:\\XIA\\Polaris\\Configuration\\default_rc.set",  
    "C:\\XIA\\Polaris\\dsp\\polaris_rc.var",  
    "C:\\XIA\\Polaris\\dsp\\polaris_rc.lst"  
};  
  
// Download to the Polaris API  
C_Handle_File_Names(boot_file_names);
```

## 2.2. C\_Boot\_System

---

### Syntax

```
long C_Boot_System (  
    long Boot_Pattern); // The boot pattern
```

### Description

Use this function to boot one Polaris board. It will first read the FPGA configurations and DSP executable code from the files stored in the host computer, and then download the configurations and code to the destination Polaris board. The final step is to download those parameters stored in the settings file to the Polaris.

Boot\_Pattern is a bit mask:

- Bit 0: Boot system FPGA
- Bit 1: Boot FIPPI
- Bit 2: Boot DSP
- Bit 3: Load DSP parameters
- Bit 4: Apply DSP parameters (call Set\_DACs and Program\_FIPPI)
- Bit 5: OFFLINE
- Bit 6: EPP\_IO
- Bit 7: USB\_IO

Under most of the circumstances, all the tasks listed in bit 0 to bit 4 should be executed to initialize the Polaris, i.e. the Boot\_Pattern should be 0x9F if using USB or 0x5F if using EPP.

### Return Values

Value	Description	Error Handling
0	Success	None
<0	Failure	Check FPGA and DSP code files; also check USB cable, etc.

### Usage Example

```
long retval;  
  
// Set up EPP port even if USB is going to be used  
// EPP address normally is either 0x378 or 0x278  
user_values[Find_User_Name("EPP_BASE")] = 0x378;  
C_User_Par_IO(user_names, user_values, "EPP_BASE", 1);  
  
// Boot Polaris using USB  
retval = C_Boot_System(0x9F);  
if(retval < 0) {  
    printf("Boot Polaris failed, retval=%d\n", retval);  
}
```

```
    return(retval);  
}  
else  
{  
    printf("Polaris has been boot up successfully\n");  
}
```

## 2.3. C\_User\_Par\_IO

---

### Syntax

```
long C_User_Par_IO (
    char *User_Par_Names[], // A two dimensional string array which
                            // contains user parameter names
    double *User_Par_Values, // A double precision array containing the
                            // User Values to be transferred
    char *User_Par_Name,    // A string variable which contains the user
                            // parameter name
    long direction );      // I/O direction (0: write, 1: read)
```

### Description

Use this function to download or upload user values between the host and the API. For those DSP I/O parameters, this function will call other utility functions to convert these parameter values to DSP recognizable values or verse versa.

### Return Values

Value	Description	Error Handling
0	Success	None

### Usage Example

```
double threshold;

// First we define User_Par_Names
char *User_Par_Names[64] = {
    "DYNAMIC_RANGE",
    "PREAMPLIFIER_GAIN",
    "PREAMPLIFIER_TYPE",
    "TRIGGER_THRESHOLD",
    "SYSTEM_GAIN",
    "BASELINE_PERCENT",
    "DETECTOR_TAU",
    "DETECTOR_POLARITY",
    "HISTOGRAM_LENGTH",
    "TRACE_LENGTH",
    "TRACE_DELAY",
    "XDT",
    "FILTER_RANGE",
    "TRIGGER_PEAKING_TIME",
    "TRIGGER_GAP_TIME",
    "ENERGY_PEAKING_TIME",
    "ENERGY_GAP_TIME",
    "PRESET_RUN_TIME",
    "PRESET_RUN_TYPE",
    "COMPTON_SHIELDING_VETO",
```

```

"EPP_BASE",
"RUN_TIME",
"LIVE_TIME",
"FASTPEAKS",
"NUMEVENTS",
"SHIELD_COUNT",
"COMPTON_COUNT",
"FTDT",
"INPUT_COUNT_RATE",
"OUTPUT_COUNT_RATE",
"PILEUP_CONTENT",
"BLCUT",
"TEMPERATURE",
"C_LIBRARY_RELEASE",
"C_LIBRARY_BUILD",
" ", " ", " ", " ", " ", " ", " ", " ",
" ", " ", " ", " ", " ", " ", " ", " ",
" ", " ", " ", " ", " ", " ", " ", " ",
" ", " ", " ", " ", " ", " ", " ", " ",
" ", " ", " ", " ", " ", " ", " ", " "
};

// Now change the Threshold to 64

threshold = 64.0;
C_User_Par_IO(User_Par_Names, &threshold, "Threshold", MOD_WRITE);

```

## 2.4. C\_Acquire\_Data

---

### Syntax

```
long C_Acquire_Data (  
    long Run_Type,                // Run type  
    unsigned int *User_data,      // Point to the data to be transferred  
    char *file_name);            // The name of the file to store data
```

### Description

Run\_Type is a bit mask whose bits 0-3 specify general run types, bits 4-7 specify actions(start/stop/poll), and bits 8-11 specify special control runs.

bits 0-3: 1 Get\_Traces()  
          2 MCA\_Run  
          3 List\_Mode\_run

bits 4-7: 1 Start new run  
          2 Resume run  
          3 Stop  
          4 Poll

bits 8-11: Special control run

Use this function to acquire ADC traces, MCA spectrum, or list mode data. The string variable file\_name needs to be specified when stopping a MCA run or list mode run in order to save the MCA spectrum data or list mode data into a file. In all other cases, file\_name can be specified as an empty string. The unsigned 32-bit integer array User\_data is used only for acquiring ADC traces (run type 0x1) or reading out MCA spectrum. In all other cases, User\_data can be any unsigned integer array with arbitrary size. Make sure that the User\_data has the correct size and data type before reading out ADC traces or MCA spectrum.

### Return Values

Value	Description	Error Handling
0	Success	None
<0	Failure	Check error message

### Usage Example

```
double user_values[64];  
unsigned int ADC_Data[8192];  
unsigned int MCA_Histogram[65536], dummy[2];  
int i, runactive, count, retval;
```

```

double timeout, poll_interval;

// Acquire ADC traces
retval = C_Acquire_Data (0x1, ADC_Data, "");
if (retval < 0) {
    // ERROR - Check error code
}

// A short MCA run

// Set desired run time
user_values[Find_User_Name("PRESET_RUN_TIME")]=10.0; // A 10-second run
C_User_Par_IO(user_names, user_values, "PRESET_RUN_TIME", 0);

// Set timeout in case the MCA run can not finish normally
timeout = 10.0; // This should not be smaller than the requested run time
poll_interval = 1.0; // Polling interval

// Start a MCA run
C_Acquire_Data(0x12, MCA_Histogram, "");

// A short delay to avoid polling too early
Sleep(100);

do {
    // Poll run status
    runactive = C_Acquire_Data(0x40, MCA_Histogram, "");

    // Wait for one poll_interval
    Sleep((unsigned long)(poll_interval * 1000));

    timeout -= poll_interval;
} while ((runactive != 0) && (timeout > 0));

// Stop the MCA run
C_Acquire_Data(0x32, dummy, "");

// Save the spectrum into a file called mca.bin; histogram data also read
// into array MCA_Histogram
C_Acquire_Data(0x52, MCA_Histogram, "mca.bin");

// 5 repeated list mode run

// initialize counter
count = 0;

do {
    // start a list mode run
    if(count == 0) // new run
        C_Acquire_Data(0x13, dummy, "");
    else // resume run
        C_Acquire_Data(0x23, dummy, "");

    // A short delay to avoid polling too early
    Sleep(100);
}

```

```
timeout = 1e6; // this can be set to any number you want, but should be
              // long enough for the run to finish normally.
do {
    // wait until run has ended (0x40 == poll)
    retval = C_Acquire_Data(0x40, dummy, "");
    timeout --;
} while ((retval != 0) && (timeout > 0));

// stop list mode run
C_Acquire_Data(0x33, dummy, "");

// Read out and save list mode data to a file called listmode.bin
C_Acquire_Data(0x53, dummy, "listmode.bin");

// increment counter
count++;

} while (count < 5); // do 5 runs
```

## 2.5. C\_Set\_Current\_Module

---

### **Syntax**

```
long C_Set_Current_Module (
    unsigned short ModuleNumber);    // The Polaris module number
```

### **Description**

Use this function to set the Polaris module number. It is mainly used in a multi-Polaris system to designate which module the host is communicating to. In a single Polaris module system, ModuleNumber should be set to 1.

### **Return Values**

Value	Description	Error Handling
0	Success	None

### **Usage Example**

```
intCardNo;

CardNo = 1;
C_Set_Current_Module (CardNo);
```

## 2.6. C\_Buffer\_IO

---

### Syntax

```
long C_Buffer_IO (  
    unsigned short *Values,           // An unsigned 16-bit integer array  
                                       // used for data transfer between the  
                                       // host and the API  
    unsigned short type,             // I/O type  
    unsigned short direction);       // Data flow direction
```

### Description

Use this function to download or upload DSP parameters between the host computer, the API and the Polaris module.

Direction: 0 (write), 1 (read)

Type: 0 (DSP I/O parameters)

1 (All DSP variables)

### Return Values

Value	Description	Error Handling
0	Success	None

### Usage Example

```
Unsigned short DSP_IO_Par[416];  
unsigned short DSP_Memory[16384];
```

```
// Read out DSP I/O parameters  
C_Buffer_IO(DSP_IO_Par, 1, 0);
```

```
// Read out all DSP parameters  
C_Buffer_IO(DSP_Memory, 1, 1);
```

## Options for Compiling POLARIS API

POLARIS API can be compiled as either a WaveMetrics Igor XOP file which is currently used by the Polaris Viewer, a dynamic link library (DLL) or static library. The two latter options can be used by advanced users to integrate Polaris into their own data acquisition systems.

The following table summarizes the required files for these options.

**Table 2.1: Options for compiling the Polaris API.**

Compile Option	Required Files		
	C source files	C header files	Library files
Standalone C-Library	boot.c, epp.c, polaris_c.c, usb.c, utilities.c	boot.h, Dlportio.h, epp.h, globals.h, Main.h, sharedfiles.h, usb.h, utilities.h	Dlportio.lib
Igor XOP	boot.c, epp.c, PolarisWinCustom.rc, polaris_c.c, polaris_iface.c, polaris_igor.c, usb.c, utilities.c	boot.h, Dlportio.h, epp.h, globals.h, Main.h, polaris_iface.h, sharedfiles.h, usb.h, utilities.h	Dlportio.lib

The Igor XOP option also needs the following files in the Igor XOP Library provided by WaveMetrics.

IgorXOP.h, VCExtraIncludes.h, Xop.h, XOPResources.h, XOPStandardHeaders.h, XOPSupport.h, XOPSupportWin.h, XOPWinMacSupport.h, XOPSupport x86.lib, and IGOR.lib.