# E. RVP8 Developer's Notes (draft)

This appendix describes the software environment that is provided to third-party developer's who wish to customize the RVP8 (and RCP8) algorithms to meet their particular needs. This information only applies to the very few organizations who have signed the *SIGMET Software Developer's License Agreement.* The vast majority of RVP8 operators and users should please skip over this entire appendix, as it is not relevant to your operational and scientific needs.

## E.1 Organization of the RDA Software

The RVP8 is SIGMET's open-architecture radar signal processor that uses standard PCI cards that can plug into any PC motherboard or embedded PCI backplane. The RVP8 software runs under standard RedHat Linux, and is developed and maintained using standard GNU tools (*gcc*, *gdb*, *make*, etc). The RVP8 therefore builds on generic high-volume PC hardware running the Linux kernel and GNU tool set. These building blocks are all mainstream, open, active technology, thus assuring the RVP8 a solid open environment for many years to come.

The larger RDA software collection, which includes the RVP8 within it, is organized into the following tree. A brief description of the contents and purpose of each directory is given:

```
/usr/sigmet/rda   Standard root point for top level of RDA tree
  * |-- intelipp     Covers for Intel Integrated Performance Primitives (IPP)
    |-- jamplayer    Altera JTAG support tools for FPGA chips on PCI cards
  * |-- kernelmod    Custom RDA kernel module to support SIGMET PCI cards
    |-- lib          Static libraries originating from within this tree
    |-- pcicards     Board-level support for RVP8/Rx, RVP8/Tx and I/O-62 cards
    |-- rcp8         Root point of RCP8
    |   |-- core       CORE RCP8 code (not delivered to customers)
  * |   |-- open       OPEN RCP8 code (available to developers)
  * |   `-- site       SITE RCP8 code (customized by developers)
  * |-- rdasubs      Common support routines for the entire RDA system
    |-- rvp8main     Root point of RVP8 main threads
    |   |-- core       CORE RVP8 code (not delivered to customers)
  * |   |-- open       OPEN RVP8 code (available to developers)
  * |   `-- site       SITE RVP8 code (customized by developers)
    |-- rvp8proc     Root point of RVP8 compute processes
    |   |-- core       CORE PROC code (not delivered to customers)
  * |   |-- open       OPEN PROC code (available to developers)
  * |   `-- site       SITE PROC code (customized by developers)
    `-- softplane    Softplane abstraction for device independent I/O

/usr/sigmet/ts   Standard root point for top level of Time Series tree
    |-- archive      tsarchive GUI code (not delivered to customers)
  * |-- archlib      Library for TS related function
    |-- exec         tsexec code, does the work of tsarchive (not delivered)
  * |-- export       tsexport and tsimport code
    |-- lib          Static libraries originating from within this tree
  * |-- switch       tsswitch code
  * |-- view         tsview code
```

All software in each of the directories marked with a "*" is provided to licensed developers.

Note that the open portions of software contain all of the hooks that scientific programmers would need to add/modify the RVP8 processing algorithms to their taste. Interestingly, this comprises only about 15% of the roughly 105 thousand lines of code that make up the complete RDA system. We do not release the "guts" of the RDA that handles memory management, low-level card drivers, PCI interrupts, PCI DMA operations, cache optimizations, on-board FPGA code, etc. In other words, the material that might unfairly be used to clone the RDA products, but which a legitimate scientific developer would not need to use.

## E.2 RVP8 Overall Code Organization

The remainder of this appendix will focus only on the RVP8 portion of the RDA development tree. The various RVP8 internal APIs gives the programmer a great deal of abstraction from the underlying hardware; and with it, freedom from worrying about things such as kernel support, interrupts, resource allocation, timing details, and the interfaces to higher layers such as IRIS and its utilities. The RVP8 is a layered software product that is organized as follows:

- ***PCI Board Firmware*** – This is the code that runs within the Field Programmable Gate Array (FPGA) chips on the SIGMET PCI cards themselves. The FPGA code is listed in this hierarchy because it is fundamental to the overall software model, i.e., *all* of the hard real-time functions of the RVP8 are implemented at the chip level on one or more PCI cards. This allows the remainder of the RVP8 to run under standard (non real-time) Linux, because no Linux process ever needs to respond to events with critically short latency. As long as there is enough average CPU time during any 500ms interval, all of the jobs will get done with no loss of data.
- ***Linux Kernel Module*** – This module is *insmod*'ed at system boot time, and provides all of the low-level PCI support for the RVP8 hardware (RVP8/Rx receiver card, RVP8/Tx transmitter card, I/O-62 card, etc). It also provides the FIFO interfaces to the IRIS DSP driver, and other services that can only be implemented at the kernel level.
- ***Card Driver Library*** – This library, along with the kernel module, constitute the low level board support package for the RVP8. Most of the hardware details remain hidden below this layer. The library is also responsible for handling FPGA firmware upgrades to the PCI boards with each release.
- ***Softplane Driver Library*** – This layer completes the hardware abstraction and provides soft configuration support for most of the electrical I/O. The *softplane.conf* file makes the association between logical control signals and physical I/O pins.
- ***RVP8/Main Threads*** – This collection of Linux threads is the foundation and support core of the RVP8, but they do not run any of the actual (I,Q) data processing algorithms. They handle all of the RVP8 configuration, setup and plotting commands, run the burst pulse analysis and AFC loop, define the system triggers and timing, and install transmit waveforms and receiver FIR filter coefficients. The Timeseries API is implemented in these threads, as is the opcode command interpreter that is documented in Chapter #6 of this manual.
- ***RVP8/Proc Processes*** – These are N-copies of identical code that are forked by the MAIN threads and which carry out all of the scientific processing (See Chapter 5) within

the RVP8. These are the actual "number crunchers" which implement most of the signal processing algorithms that operate on raw (I,Q) data. The code is written in standard C, but uses a set of optimized Pentium/Athlon library functions for floating point FFT, convolution, filtering, dot product, etc. By calling these very fast primitives, ordinary C code is adequate for programming the processing algorithms. This makes the code very maintainable. Generally you will create one RVP8/Proc process per available CPU on an SMP machine. Most of your customization of the RVP8 algorithms will likely occur in these processes.

Virtually all of the RVP8/Proc code is released to licensed developers as source code. Likewise, there are open APIs for the RVP8/Main that allow you to build custom trigger patterns, phase sequences, etc., on the transmitter side. Finally, the kernel source is available so that developers can run within their own customized Linux environment.

## E.2.1 RVP8 Software Maintenance Model

The remarkable thing about SIGMET's open source developer's model is that the open code *is* the actual delivered code. It is neither example code, nor an abridged form of the final delivered algorithms. When we build an official RDA release we're using the exact same rvp8main/open and rvp8proc/open source directories that our licensed developers would receive. All bug fixes, enhancements, new processing modes, and maintenance work will always be reflected directly in these source files -- there is no "other" version. This point is crucial in the model for software maintenance that includes both SIGMET code and customer code.

The CORE portion of the developer's tree contains no source files, and is delivered only as compiled binaries, whereas the OPEN portion of the tree is fully populated with the actual source files used by SIGMET to build each RVP8 release. This OPEN code should be used only as a reference for programming examples and ideas; *it should never be directly modified by the developer.* The reason this is so important is that the OPEN code may change significantly with each new release. If custom changes were made in this area, those changes would all be lost when the next RDA update was installed.

The correct way to add custom features to the RVP8 is to add them to the SITE directory. This directory is delivered simply as a collection of empty program stubs whose calling conventions are very stable. Each stub is basically a hook that can be used to define an entirely new RVP8 major mode. By replacing these stubs with custom code, custom major modes can be added to the RVP8. And since the delivered SITE stubs are simply thrown away and replaced with their customized versions, the long term maintenance of both the SIGMET portions and developer portions of the RVP8 code are assured.

## E.2.2 Installing Incremental RDA Upgrades

RDA development systems are upgraded in the same manner as standard operational systems using the procedures described in the *Software Installation Manual*. The only difference is that you have a little more work to do afterward to continue running your custom code under the new RDA release.

The RDA upgrade will replace the RVP8/Main and RVP8/Proc SITE libraries with their default stubs, so you will need to rebuild your customizations. Since the executables are built from the OPEN trees, the simplest method is to copy those newly released OPEN areas alongside your existing SITE directories. Your entire development cycle can then be carried out entirely from the */home/operator* area, without ever having to compile in the */usr/sigmet* tree directly.

```
$ cd /home/operator/sigmet/rda
$ rm –rf rvp8main/open rvp8main/core rvp8proc/open rvp8proc/core
$ cp –pr /usr/sigmet/rda/rvp8main/open rvp8main
$ cp –pr /usr/sigmet/rda/rvp8proc/open rvp8proc
$ compall cleanexe ; compall –j2 install
```

What's important here is that you are rebuilding your code under the new header files that were just installed during the upgrade. As such, you are likely to find a few minor incompatibilities in the form of changed structure definitions and/or changed function prototypes. Check the header file documentation as well as the current *RDA Release Notes* to find out whatever (usually simple) changes need to be made. The RVP8 will not let you run with incompatible headers, producing RVP8/Main startup errors such as:

```
RVP8/Main code version mismatch
  Core: Version=2.12  Build=298
  Open: Version=2.12  Build=298
  Site: Version=2.11  Build=295
```

and RVP8/Proc errors later in the initializations such as:

```
Forking parallel compute process(es)...
  RVP8/Proc–0: Requesting exit due to signal 1
  RVP8/Main: UNIX Signal: Unexpected RVP8/Proc termination
  RVP8/Main: RVP8/Proc version mismatch <ProcSite: Ver=2.11 Bld=295>
```

## E.2.3 Rebuilding the RDA Linux Kernel Module

The RVP8 requires kernel driver support that is provided with each RDA release in files whose names resemble */usr/sigmet/bin/rda/rda–2.4.20–6smp.o*. If you are running on standard RDA hardware, the *rdasys start* script will find the appropriate kernel module for your system and install it automatically. However, if you are running with a custom Linux kernel, you will have to rebuild your RDA kernel module whenever its version has changed since the last time that you built it. The RVP8 will not run with incompatible modules and will report an error:

```
RVP8/Main: KernelMod mismatch
  <Found SIGMET RDA kernel module V3.3, but V3.2 is required>
```

Before building the RDA kernel module for the first time, you'll need to prepare the kernel source tree. This only needs to be done once as *root* :

- Make a symbolic link */usr/src/linux —> /usr/src/linux–2.4.18–3* (or whichever version of kernel source you are running)
- Do a *make xconfig* in /usr/src/linux to startup the interactive configurator.
- Choose *load configuration* from the "file" menu and enter the path and filename for the specific kernel you are running. **Do this only if you do not have an existing *.config* file already**, lest you clobber the previous work of others.

- Under *kernel hacking* set "kernel debug" to "off"
- Save your changes
- Edit */usr/src/linux/Makefile* and change the EXTRAVERSION line to match your currently running kernel, for example "–10smp"
- In /usr/src/linux run "make dep" to build your dependencies.

Then, exit from the *root* shell and rebuild the RDA module based on the kernel headers that reside in the kernel source area. Again, this is generally the only step that you'll need to do for each incremental RDA release; and even then, only when the RVP8 mismatch startup error is reported.

```
$ cd /usr/sigmet/rda/kernelmod
$ compall clean
$ compall –j2 install
```

The new module will be automatically loaded on the next boot, or it can be manually installed right now (again, as *root* ) using:

```
# rdasys stop
# rdasys start
```

## E.3 Debugging and Profiling Your Code

Although the complete RVP8 is a rather complex multi-thread and multi-process system, it is still "merely" a user-level application running under the Linux operating system. As such, most of the of the custom code that you develop can be debugged using tools that are already familiar to Linux/C/GNU programmers. This section presents an overview of debugging and monitoring techniques that are available to RVP8 developers.

### E.3.1 Monitoring Opcode/Data Activity: *–exposeIO*

The RVP8 is generally controlled by some higher level application such as *ascope*, *iris*, *dspx*, etc., which communicates with the RVP8 via the *IRIS DSP Driver Library* using the opcodes described in Chapter 6. These layered applications provide a clean and maintainable signal processor interface, as evidenced by the RVP5/6/7/8 being largely opcode-compatible over a fifteen year period.

The complete opcode activity between the application driver and the RVP8 can be viewed by including the *–exposeIO* flag on the RVP8 startup command line. The following printout shows what happens when an "Output Test" opcode is written, followed by a "Noise Sample" command and a "Get Processor Parameters" opcode:

```
Opcode 0x0004 (OTEST)
Output Words
    0: 0001 0002 0004 0008  0010 0020 0040 0080  0100 0200 0400 0800  1000 2000 4000 8000
Opcode 0x0005 (SNOISE)
Input Words
    0: 0000 0000
Opcode 0x0009 (GPARM)
Output Words
    0: 2000 0064 0960 9DCF  0110 0DD0 0000 0000  0000 5284 0000 0000  0040 D472 0000 0000
   16: 0000 0603 020D 5DC0  012C 1D4C 1770 0BB8  8421 0210 2EE0 2EE0  0000 0000 042E 07AE
   32: 000D FE20 0066 0050  FE70 0000 0000 0000  0000 0000 0001 0011  0000 0011 0000 0DD0
   48: 0000 0000 00E8 01C0  03E8 04B0 0303 0000  0037 30CB 0003 0000  0000 0000 0000 0000
```

When the OTEST opcode is received it is parsed as such by the RVP8/Main *HostCmds* thread, which also generates the *exposeIO* printout. The opcode is shown in both numerical and mnemonic form, followed by the sixteen-word walking-ones pattern that results from it. The SNOISE command then arrives, along with the two input words which are expected by that opcode. No output words are generated by SNOISE. Finally, the GPARM opcode is received and the 64 output words that it produces are displayed.

Being able to watch the opcode-level activity of the signal processor can be very helpful when debugging both custom driver code and custom opcode handlers that you might write for the RVP8.

## E.3.2 Showing Live Acquired Pulse Info: *–showAQ*

The (I,Q) data that are computed by the FIR filters on the RVP8/Rx PCI card are transferred to CPU memory by way of DMA (Direct Memory Access) bus cycles that are initiated by the RVP8/Main threads. This is an implementation detail that developer's can largely ignore, except to be aware that the radar data always arrive in discrete "chunks" and that the TimeSeries API is updated accordingly. The following printout shows the (I,Q) bus activity for a Dual-Polarization (dual RVP8/Rx cards) system when the *–showAQ* flag is included on the RVP8 startup command line.

```
AQ:193 pulses(1:1B96 – 1:1C57) 32038 words(8:2762F – 8:2F355) 157ms
AQ:192 pulses(1:1B9B – 1:1C5B) 31872 words(8:27B0A – 8:2F78A)   3ms
AQ:191 pulses(1:1C57 – 1:1D16) 31706 words(8:2F355 – 8:36F2F) 157ms
AQ:192 pulses(1:1C5B – 1:1D1B) 31872 words(8:2F78A – 8:3740A)   4ms
```

Here we see 193 pulses of packed 32-bit (I,Q) data words being transferred from the first Rx card, followed 3ms later by a similar transfer from the second Rx card. Each block of data is moved directly into the virtual address space of the RVP8/Main without requiring any cycles from the CPU itself. Then, a DMA-Done interrupt causes the *IQ-Data* thread to wakeup and unpack that card data into FLT4 values which are written to the TimeSeries API.

Don't worry too much about the other numbers that are printed from *–showAQ*. The flag's main use is to show how chunks of timeseries data are being made available to the RVP8 processing threads. If you're curious about the timing details, this is a simple way to take a look.

## E.3.3 Showing Coherent Processing Intervals: *–showCPIs*

Coherent Processing Intervals (CPI's) are blocks of acquired pulses that have been selected as the input data for the computation of each processed ray. You can monitor how the timeseries data stream is being organized into rays by supplying the *–showCPIs* flag on the RVP8 startup command line. A sample printout while running *ascope* is shown below.

```
CPI 0: 64–Pulses (269.74, 1.49) to (271.25, 1.49)  126.00–ms  TS: 1%
CPI 1: 64–Pulses (270.51, 1.49) to (272.02, 1.49)  126.00–ms  TS: 0%
CPI 2: 64–Pulses (271.28, 1.49) to (272.79, 1.49)  126.00–ms  TS: 0%
CPI 3: 64–Pulses (272.04, 1.49) to (273.56, 1.49)  126.00–ms  TS: 1%
```

The CPIs are numbered beginning with each new ProcSection, and the number of pulses within each one is shown. The starting (AZ,EL) and ending (AZ,EL) are then printed, along with the duration of the CPI in milliseconds. In the above example the PRF was 500Hz, hence, the 64 pulses span a total time of 126ms. Finally, the "lateness" of the data being extracted from the TimeSeries API is listed as a percentage of available history depth.

If the lateness approaches 100%, that is evidence that the RVP8 is having trouble keeping up with the CPU and/or memory throughput demanded by the current major mode. The default algorithm for blocking CPIs keeps track of whether it seems to be falling behind, and tries to gracefully skip pulses in order to catch up. Custom CPI algorithms for intensive major modes should be written with the same sort of feedback. Please see the code in *rvp8main/open/cpi.c* for more details and suggestions.

## E.3.4 Showing RealTime Callback Timers: *–showRTCtrl*

The RVP8 can show the detailed activity of whatever callback timers have been registered for the current major mode. If the *–showRTCtrl* flag is supplied on the command line, then timer activity will be printed during each active ProcSection. These timer callbacks provide a simple yet powerful framework for developers to handle real–time events within the RVP8 (See Section E.5). Please also see the documentation for *struct rtCtrlCBTimer.*

When no callbacks are registered, or when the registered callback does not request any further activity, the printout will simply look like this:

```
RTC  –First–  #0(–)  #1(–)  #2(–)
RTC Disabling further callbacks for this PROC section
```

The first real–time callback does not set any of the *iTVWaitOnNext* fields, and the whole timer mechanism simply goes to sleep until the next ProcSection begins. In contrast, the following printout was generated by the demonstration histogram callback that is provided in *rvp8main/open/rtctrl.c* :

```
RTC  –First–  #0( dV:0 F:0 Rq:100000 )  #1(–)  #2(–)
RTC Setting timer #0 clock source to 0 (1MHz)
RTC 0.100068  #0( dV:100009 F:1 Rq:2500 )  #1( dV:0 F:0 Rq:5 )  #2(–)
RTC Setting timer #1 clock source to 1 (Trig)
RTC 0.002512  #0( dV:2513 F:1 Rq:2500 )  #1( dV:2 F:0 Rq:5 )  #2(–)
RTC 0.002510  #0( dV:2510 F:1 Rq:2500 )  #1( dV:3 F:0 Rq:5 )  #2(–)
```

Here we see the first invocation requests a callback in 100000 counts of the 1MHz timer. The "Rq" (request) field of timer #0 shows the *iTimerWait* duration, and the *iTVWaitOnNext* field causes the 1MHz clock to be selected as the timer source.

That callback fires a little more than a tenth of a second later (the additional 68µsec represents the Linux Interrupt-to-Running latency, plus the time to set the clock source in the first place). On the second invocation, the demo callback requests a delay of 2500 on the 1MHz timer (2.5ms), and a delay of five counts on the Trigger timer. We also see the RVP8 trigger being selected as the clock source for timer #1. Note that the clock source messages are printed only when changes are made.

From then on the callbacks fire regularly based on activity on timers #0 and #1. The "dV" number show the "delta-value" for each timer, i.e., the change in the timer's count since the previous call. The "F" field indicates whether each timer has fired (1) or is still counting up to the requested delay (0). In this case, timer #0 is regularly firing every 2.5ms; and since we only get two or three trigger counts in that much time, timer #1 never actually fires at all. If the PRF were increased, however, we would suddenly see timer #1 counting up to five triggers and then firing before the 2.5ms expires.

## E.3.5 Using *ddd* on the Main & Proc Code

The GNU *ddd* symbolic debugger is (usually) built on top of the *dde* command line debugger. Both are mature and remarkably well crafted tools that are provided on all Linux systems.

Code that you write for the RVP8/Main threads can be debugged simply using:

```
$ cd /home/operator/rda/rvp8main
$ compall –j2 install
$ cd open
$ ddd rvp8
```

Here the SITE and OPEN code is first compiled in the private development tree, and we then run the RVP8 executable that resides in the OPEN area. You may want to redefine the environment variable "OPTIMIZEFLAG=–g", i.e., remove the "–O2" that is normally there. This will cause the Makefiles to build non-optimized code which generally behaves more smoothly in a symbolic debugger.

You can simply type "run" in the *ddd* execution window, but you may prefer to use "run –nod" to skip the powerup diagnostics and speed up your development cycle. Typing "b main" ahead of time will cause *ddd* to break on the first executable line after all of the dynamic library references have been resolved. This is usually a good way to get started because you may then break on any entry point within the RVP8. Before the libraries are loaded, it is possible that *ddd* may not be able to find all of its symbols.

**Note: When the RVP8/Main is debugged in the above manner, signal events originating from *ddd* will also be sent to the RVP8/Proc child threads and are likely to cause them to behave improperly. Use this technique only for quick ventures into the main threads, and preferably with *–procs 0*. Please read on.**

Debugging the RVP8/Proc code is similar, except that we do not want it to be automatically started from the main threads. Therefore, in one X-Term window type:

```
$ rvp8 –noFork
```

which will startup the RVP8/Main threads, but pause at the point that the RVP8/Proc process would normally be created. Note that *–noFork* forces the subprocess count to one, as if you had included *–procs 1* on the original command line. Then, in another window type:

```
$ cd /home/operator/rda/rvp8proc
$ compall –j2 install
$ cd open
$ ddd rvp8proc
```

which builds the new RVP8/Proc code and starts up the debugger. Typing "run" in *ddd* will startup the subprocess; and when it has finished its initializations, the RVP8/Main will magically continue in the other window.

**Note: The proper way to debug the RVP8/Main builds on this technique. Run *ddd rvp8* as described above, but include the *–noFork* flag in *ddd's* "run" command. Then run rvp8proc manually in another window, either with or without its own *ddd*. You can create two simultaneous Main and Proc *ddd* sessions in this manner, and signals from one will not interfere with the other.**

## E.3.6 Finding memory leaks with *valgrind*

The *valgrind* profiler can be useful if you are having runtime problems that are hard to track down. The most common problem that it solves is finding reads-before-writes, i.e., when you forget to set a value in a structure somewhere, and then reference it later before actually writing into it. Malloc/Free inconsistencies are also easily diagnosed with *valgrind*.

The latest version can be downloaded from **http://valgrind.kde.org**. *Valgrind* is very easy to use because you simply run it on the executable being debugged in the same ways that *ddd* was used in the previous section. Moreover, there are no special compiling or linking requirements. To debug the RVP8/Proc process, for example, simply use:

```
$ cd /usr/sigmet/rda/rvp8proc/open
$ valgrind –-tool=memcheck rvp8proc
```

## E.3.7 Profiling with *gprof*

The GNU tools include the handy runtime profiler *gprof*. This tool works in conjunction with the C-compiler, and analyzes statistics in the *gmon.out* file that is produced when the running program exits. The RDA Makefiles are already setup to build profiled versions of either the RVP8/Main or RVP8/Proc executables. To do this, define ***one*** of the following environment variables:

```
export PROFILE_RVP8MAIN="1"
export PROFILE_RVP8PROC="1"
```

Then do a "make clean" and "make install" in the SITE and OPEN portions of the relevant tree. If you are profiling the RVP8 compute process you'll need to make sure that only one of them is forked by including "–procs 1" in the original startup command. Otherwise, each sub-process will attempt to create the same *gmon.out* and chaos will surely follow. If you forget to specify the solitary process option, the RVP8 will force it upon you anyway but with an accompanying warning message. Likewise, the RVP8 will not let you profile both the Main and Proc executables at the same time.

Since the *rvp8* and *rvp8proc* executables are built from their OPEN directories, running the profile analysis from within that directory will allow *gprof* to find its symbols, e.g.,

```
$ cd /home/operator/sigmet/rda/rvp8proc/open
<Run the RVP8 for a while...>
$gprof rvp8proc –I ../site
```

## E.4 Creating New Major Modes from Old Ones

Custom algorithms are added to the RVP8 by building on its concept of Major Modes and Output Data Types. Each Major Mode defines all of the methods that are required to compute each of the Output Data Types from raw (I,Q) data. Therefore, by allowing users to define their own Major Modes, one has all of the hooks required for full customization. You can code up your own custom algorithms by making incremental changes to one of the SIGMET models; or you can start from scratch and build something completely unique.

The best way to create a custom major mode is usually to start with the code for an existing one and incrementally modify it to include the new features. The FFT mode, for example, is defined in the files *rvp8main/open/mt_fft.c* and *rvp8proc/open/ct_fft.c* , each of which contains only about 100 lines of boilerplate toplevel definitions. Creating your new major mode would likely begin by copying this prototype code into separately named files in the *rvp8main/site* and *rvp8proc/site* areas, and then proceeding to make the desired changes.

There are four major mode slots reserved for custom user applications. The names of your new modes are defined using the **setup** utility's *RVP–>Optional Data Parameters* area. Names can be up to fifteen characters long, and whatever text you choose here will automatically appear later in pulldown menus for **ascope**, **iris**, etc. Your new code is invoked by modifying one of the lines of *rvp8main/site/mt_user.c* and *rvp8proc/site/ct_user.c* . These files are very simple and are shipped in deactivated form as follows:

```
/* ------------------------------
 * The available user modes are shipped in an unused state.  By doing
 * nothing in their INIT routines, these modes are effectively
 * disabled (all function pointers remain NULL).
 */
void mtInitMajorMode_user1( void ) {}
void mtInitMajorMode_user2( void ) {}
void mtInitMajorMode_user3( void ) {}
void mtInitMajorMode_user4( void ) {}
```

All that's required for execution is that you call your custom initialization routine(s) from one of these predefined user stubs:

```
void mtInitMajorMode_user1( void ) { initMajorMode_myCustomMode() ; }
```

### E.4.1 Function Pointers are the Key to Customization

Each major mode is characterized by a set of function pointers or *methods* which define how certain critical operations are to be carried out. The main RVP8 threads are governed by the following methods:

```
struct rvp8MainMajorMode {      /* Customized processing routines */
           privateData_t *privateData          ;
         exitMajorMode_f *exitMajorMode        ;
       initProcSection_f *initProcSection      ;
       exitProcSection_f *exitProcSection      ;
            rtCtrlCBF_f *rtCtrlCBF             ;
  iNominalTrigSequence_f *iNominalTrigSequence ;
```

```
      iCustomTrigSequence_f *iCustomTrigSequence  ;
         customUserOpcode_f *customUserOpcode     ;
     cwpulseMatchedFilter_f *cwpulseMatchedFilter ;
        iTxWaveformDesign_f *iTxWaveformDesign    ;
      rawPulseCorrections_f *rawPulseCorrections  ;
      rawPulseCorrections_f *targetSimulator      ;
             lConfigError_f *lConfigError         ;
             lFindNextCPI_f *lFindNextCPI         ;
              lCheckupCPI_f *lCheckupCPI          ;
        lAssignProcsInCPI_f *lAssignProcsInCPI    ;
            setupProcBins_f *setupProcBins        ;
       lAnalyzeBurstPhseq_f *lAnalyzeBurstPhseq   ;
          getAzElPosBtime_f *getAzElPosBtime      ;
        sampleLiveAzElPos_f *sampleLiveAzElPos    ;
        insertLiveAzElPos_f *insertLiveAzElPos    ;
        frontPanelDisplay_f *frontPanelDisplay    ;
} ;
```

User's of object-oriented languages will recognize this sort of structure as allowing new objects to inherit existing properties of old objects, while still permitting individual "personality" to enter as needed. Like the Linux Kernel, the RVP8 is written in standard "C" and uses function pointer replacement to accomplish customization. Much of the RVP8's code organization was patterned after the elegant and far more intricate implementation of driver module replacement within Linux.

Note that the *initMajorMode()* routine which creates each major mode in the first place is not part of this list because it is separately invoked (See Section E.4) in order to first establish the list. Detailed descriptions of the various methods are included along with their definitions in *rvp8main.h* and *rvp8proc.h* . A few are so fundamental that they're worth repeating here:

**`privateData_t`** : Each major mode can allocate a private data area for whatever memory resources are required during the operation of that mode. This private area is created and prepared by the *initMajorMode()* routine as part of its filling in the entire list of methods at the start of each major mode. The *exitMajorMode()* handler is responsible for releasing all private memory resources.

**`exitMajorMode_f`** : This routine releases all resources that were allocated during the initialization and execution of the major mode. The EXIT routine is called whenever the major mode changes, or whenever we are "between" modes, e.g., after a TTY Chat "q" command, or a processor reset. The RVP8 is guaranteed not to be within an active PROC Section (See below) when *exitMajorMode()* is called; i.e., if *exitProcSection()* needs to be invoked, that will always happen first.

**`initProcSection_f`** & **`exitProcSection_f`** : Pair of routines that will be called whenever the RVP8 enters and exits active timeseries acquisition and processing. The INIT routine is called when a PROC opcode is executed that causes the *IQData* thread to begin collecting data, and activates the RVP8 compute threads for the current major mode. The EXIT routine is called whenever anything interrupts those processes, e.g., the execution of most other opcodes. Note that the major mode itself does not change at these transitions; rather, these functions allow the current major mode to interact with the timeseries data acquisition as needed.

# E.5 Real-Time Control of the RVP8

The RVP8/Main includes a dedicated thread (named "*RT-Ctrl*") which can be programmed to handle realtime events while the processor is running. This includes activities such as altering trigger patterns in response to antenna position or other external conditions, tracking live inputs from an I/O-62 card, etc. The *RT-Ctrl* thread runs at a priority that is higher than any other RVP8 thread. This allows it to preempt other activities to achieve near realtime behavior; but as such, it should always be coded as efficiently as possible and should not do anything that could possibly become CPU bound.

Despite the high POSIX static priority of *RT-Ctrl*, its scheduling is still subject to jitter due to uncertain latencies within the Linux kernel. The magnitude of this jitter will depend on the kernel version, the type of motherboard being used, and the nature of the other processes that are competing for CPU time. We've found that intense disk and network I/O are among the worst contributors to high scheduling latency, sometimes amounting to as much as 25ms of variation. Fortunately, these uncertainties can usually be absorbed by proper coding of the thread.

## E.5.1 Using the Programmable Callback Timers

The *RT-Ctrl* thread is structured around a flexible set of realtime callback timers. The thread is activated each time the *initProcSection* method is called, and it is deactivated when that PROC section is eventually exited. Thus, realtime control is available whenever live (I,Q) data are also being acquired and processed.

The **rtCtrlCBF_f** callback function is customized to handle the realtime control and sequencing tasks for the current major mode. This routine will be called once at the beginning of each data processing section. It then performs whatever work needs to be done, and requests that it be called back at some later time. Up to three independent timing and/or event criteria can define when the callback will occur, and each can be based on:

- A specified number of counts of a freerunning 1MHz counter (clock time)
- A specified number of trigger pulses (trigger relative time)
- A specified number of external input line transitions (I/O event time)

The callback will occur as soon as the timeout criteria are met for any of the three timers that are activated. For example, if Timer-0 requests a callback after five triggers, and Timer-1 requests service after 10000 1MHz counts, then at 1KHz PRF the callback will occur in 5ms. Note that the callback sequence can be terminated at any time within the current PROC section simply by specifying void criteria for reentry.

Callback routines can request that a new trigger bank be loaded at the precise instant that the next timer event(s) fire. This special feature is implemented in hardware on the RVP8/Rx card, and is necessary for creating alternating trigger patterns that remain completely unaffected by Linux interrupt latencies. Precise programmable trigger control is an important application of the *RT-Ctrl* thread, and is made possible by having this hardware support at the PCI card level.

## E.5.2 Example: Standard Trigger/Antenna Events

Please refer to the source file *rvp8main/open/rtctrl.c* which contains the standard RVP8 code for live trigger control and angle synchronization. The software consists of these pieces:

**initProcSection_dflt** – This is the default routine for initial entry into each PROC section, and its primary job is deciding what kind of realtime callback responses are needed for the current RVP8 configuration. It sets up an area of private memory that will later control the realtime activity (see calling example in *rvp8main/open/mt_fft.c* ).

**rtCtrlCBF_dflt** – This is the default realtime callback of the *RT-Ctrl* thread. It receives a subset of the private memory for this PROC section that was set aside for realtime control (see calling example in *rvp8main/open/mt_fft.c* ), and performs one of the following:

- **Angle synchronization** – A brief history of prior antenna angles are entered into a least squares model, which is then used to predict how long it will be until the next sync angle boundary is crossed. A callback based on the 1MHz counter is then requested for that time using one of the timers:

  ```
  cbt->iTVWaitOnNext = RTCBTV_1MHZ ;
  cbt->iTimerWait = (1000 * iFutureMS) ;
  ```

  along with an immediate hardware assisted trigger bank change using:

  ```
  cbi_a->iTgBank = iBankNew ; cbt->lTgBankAutoStart = TRUE ;
  ```

- **Freerunning Dual-PRF** – This mode is much simpler, and simply alters the pattern of triggers after a computed number of pulses have occurred:

  ```
  cbt->iTVWaitOnNext = RTCBTV_TRIG ;
  cbt->iTimerWait  = ceil( MAX( 1.0, fTrigs ) - 1E-4 ) ;
  cbt->iTimerWait -= cbt->iTimerError ;
  ```

Note that interrupt latencies are absorbed differently in these two cases. For angle sync'ing we compute the expected crossing time based on where the antenna happens to be when the callback was entered. It does not matter if a callback is delayed, because we'll simply compute a shorter future time in such cases. For Dual-PRF triggers we likewise need to shorten the next interrupt whenever the present callback is delayed, but we do this using the *iTimerError* field that keeps track of how late (measured in timer events) we presently are.

## E.5.3 Example: RealTime Interrupt Histogram

Please refer to the source file *rvp8main/site/demohist.c* which contains demonstration code for a realtime callback that prints a histogram showing the scatter of callback times brought about by Linux scheduling latencies. First, arrange for this code to be attached to a user major mode by editing *rvp8main/site/mt_user.c* :

- Change the default USER1 major mode init routine to call the demo histogram:
  ```
  void mtInitMajorMode_user1( void ) { initMajorMode_demohist() ; }
  ```
- Include the header file consisting of a single line prototype for the above function:
  ```
  #include "demohist.h"
  ```
- Compile, install, and run your changes with:
  ```
  $ cd /usr/sigmet/rda/rvp8main
  $ compall cleanexe ; compall -j2 install
  $ rvp8 -noDiags
  ```

The RVP8 will startup normally. Next, run the **ascope** utility and request USER1 major mode from the *Gen Setup* menu. You should then see messages printed in the RVP8 startup X-Term indicating that the major mode has been entered, and that the PROC section has been initialized:

```
Beginning PROC section demohist code.
  Target scheduling interval is 2.500 ms OR 5 triggers
  Will print timing jitter histogram every 10 seconds
```

The realtime callback handler is now collecting statistics on its interrupt times, and is scheduled every 2.5 milliseconds or every five triggers, whichever is fastest. Moreover, a histogram will be printed every ten seconds showing the distribution of times. Some interesting things to try:

- Type a low PRF (~500Hz) into **ascope** and verify that the interrupts cluster around 2.5ms. Then type a high PRF (~5KHz) and verify that histogram peaks at 1ms.
- While the callback handler is running, make the RVP8 machine busy in some manner, e.g., by running compilers, reading large files, etc. You should see noticeable scatter of the histogram plot as other processes compete for CPU scheduling.

Lastly, verify that selecting some other major mode in **ascope** results in the messages:

```
Exiting from demohist PROC section
Exiting DemoHist major mode
```

## E.6 Customizing the (I,Q) Data Stream

### E.6.1 Defining the FIR Matched Filter

### E.6.2 Applying Raw Pulse Corrections

### E.6.3 Inserting *UserIQ* Header BIts

## E.7 Customizing the Front Panel Display

## E.8 Adding Custom DSP/Lib Opcodes

## E.9 Using the Softplane for Physical I/O

### E.9.1 Softplane Programmer's Model

### E.9.2 Reducing Unnecessary PCI Traffic

## E.10 Handling Live Antenna Angles

## E.11 Creating Custom Trigger Sequences

### E.11.1 Defining Trigger Waveshapes

### E.11.2 Defining Trigger PRT Sequences

### E.11.3 Polarization and Phase Control

### E.11.4 Example: Adding PRT Micro-Stagger

Please refer to the source file *rvp8main/site/demostag.c* which contains demonstration code for building custom trigger timing within a new major mode. A special "micro-stagger" trigger generator is implemented in which the trigger PRT varies a tiny amount (a few microseconds) from pulse to pulse. This causes multiple trip echoes to become incoherent when viewed relative to the first trip Tx phase, thus providing a very simple method of "whitening" the Doppler signature from range aliased echoes.

The code consists of two parts:

- **initMajorMode_stag** – The major mode initialization routine is a direct clone of the FFT code from *rvp8main/open/mt_fft.c* . The only difference is that the default trigger generation is superceded by the custom *iNominalTrigSequence_stag* method.
- **iNominalTrigSequence_stag** – This is a direct clone of the factory default trigger code in *rvp8main/open/txsubs.c* , except that the PRT sequence is altered by a zero-mean pseudo-random set of time staggers.

To compile and run this mode, please modify *rvp8main/site/mt_user.c* as described in Section E.5.3. In addition, we'll want to import the default FFT behavior for the compute processes by changing *rvp8proc/site/ct_user.c* to:

```
void ctInitMajorMode_user1( void ) { ctInitMajorMode_fft() ; }
```

Build and install all of these changes by compiling both the *rvp8main* and *rvp8proc* trees. You may then request this user mode from **ascope** and verify the micro-staggered PRTs on a delayed sweep oscilloscope. For example, at 1KHz PRF and with a delayed sweep of 1ms, you should see a flurry of "second bang" triggers whose leading edges jitter over a 5μsec span. Contrast this with the normal FFT mode which has constant interpulse spacing.

Another way to verify the staggered PRTs is to check the live timeseries data using the example source utility that is included in the *rdasubs* directory:

```
$ /usr/sigmet/rda/rdasubs/rvp8ts_example –headers
–SeqNum– ––AZ–– ––EL–– PrevPRT NextPRT Bank Wave TX
00014289   0.00   0.00  720088  720120    0   60 00
0001428A   0.00   0.00  720120  720080    0   61 00
0001428B   0.00   0.00  720080  720056    0   62 00
0001428C   0.00   0.00  720056  719952    0   63 00
0001428D   0.00   0.00  719952  719992    0    0 00
0001428E   0.00   0.00  719992  719832    0    1 00
0001428F   0.00   0.00  719832  720040    0    2 00
```

The PRF was set to a low value of 100Hz in order to limit the rate at which the live headers were printed. A 72MHz IFD was used in this measurement, hence the nominal time to the previous and next 100Hz pulses would be 720000 clock ticks. However, because of the micro-stagger, the actual interpulse PRTs are seen to vary.

## E.12 Determining CPI's and Ray Boundaries

## E.13 Using the RVP8 TimeSeries API

The RVP8 TimeSeries API is the fundamental interface through which (I,Q) data are made available to all application code which requires them. This API is central to the design and operation of the RVP8 itself, and is used by the parallel compute processes to access incoming timeseries data. Because the API is used so heavily, the entry points have been reasonably stable and well debugged since the early days of the RVP8.

The TimeSeries API is provided within a larger collection of RDA support services that are located in *rda/rdasubs*, with the defining header file *include/rdasubs_lib.h* . Please see the documentation in that header file for the most recent API definitions.

### E.13.1 Reader and Writer Clients

The timeseries API is entirely stateless and passive from a reader client's point of view, i.e., it allows any number of callers to eavesdrop on the (I,Q) data as they arrive, but there are no control actions passed back in the other direction. The reason that an event driven model is not provided is that the API is fundamentally a single–writer / multiple–reader interface. There is no private state maintained for each reader client that hooks up to it. This was a design goal from the start; readers should be able to come and go willy-nilly without affecting anything else. As such, the notion of 'notify me when new data are available' would not be well defined, because 'new' would have to be a per-client notion, i.e., 'new' since the last data that each particular client happened to look at.

Also, the API is really most valuable in providing random access to the recent buffered (I,Q) data. Because of the PCI/DMA buffering this is a pseudo realtime interface, and the data are typically 100–400ms delayed from their actual time of arrival. As such, it is not terribly important to be able to track the leading edge of newly arriving data with any particular precision. There are about two seconds of data buffered within the timeseries API; so merely checking at 10–20Hz presents a negligible CPU load, does not risk losing any data, and matches the PCI/DMA burst transfers.

### E.13.2 Attach/Detach Details

### E.13.3 Extracting Pulses via Sequence Numbers

### E.13.4 Using Memory Bandwidth Effectively

### E.13.5 Packed and Floating Data Formats

## E.14 Using the Intel IPP Library

The IPP software enables taking advantage of the parallelism of the single-instruction, multiple-data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions. These technologies can vastly improve the performance of computation intensive signal processing applications. Please refer to the complete *Integrated Performance Primitives Reference Manual Volume 1* which is supplied in PDF form on each RDA CDROM.

The data types used within the IPP library are, for the most part, compatible with the standard SIGMET typedefs; hence, the IPP routines can almost always be called directly from RVP8 code. You can find many examples of this simply by *grep*'ing the RVP8 sources for "ipps". Status codes from the IPP library can be converted into RDA MESSAGEs via *sigIppStatus()*.

The SIGMET header file *intelipp_lib.h* should be used to define the IPP library entry points. Do not include the Intel headers directly because doing so will bypass some RDA consistency checks and will make your code less maintainable.

If you are new to the Intel IPP library we provide an example file *rda/intelipp/ipp_example.c* that you can read through to get started. It allows you to run DFT's and matrix transposes of various sizes, and provides a simple method of benchmarking the IPP library on your hardware.

The IPP runtime libraries are bundled into each RDA release. SIGMET has purchased a single-user developer's license from Intel which allows one engineer to develop code which links to the IPP library, and then to distribute an unlimited number of copies of that code in executable form. Our license does not, however, allow for any additional programmers to develop their own code as an extension of SIGMET's license.

Essentially, the IPP license is a per-developer license. There are no restrictions or royalties on the distribution of compiled binaries, but each additional developer must be licensed at a cost of approximately $250/year. Some relevant text is included below from the Intel online FAQ **http://www.intel.com/software/products/ipp/faq_lic.htm** . The bottom line is that our community of RDA developers must individually license themselves with Intel in order to write new code that uses the IPP library.

- ***What are the redistributable files?***
  In general, the redistributable files include the linkable files (.DLL and .LIB files for Windows*, .SO files for Linux*) including the Runtime Installer. With your purchase of the Intel IPP product (and updates through the support service subscription), the redist.txt file outlines the list of files that can be redistributed.

- ***Do I need to buy an the Intel IPP license for each copy of our software that we sell?***
  No. there is no per copy royalty fee. Please check the Intel IPP end user license agreement for more details.
- ***How many copies of my company's application can redistribute the Intel IPP library files?***
  You may redistribute an unlimited number of copies of the files that are found in the directories defined in the Redistributables section of the end–user license agreement.
- ***Are there royalty fees in using the Intel IPP?***
  No, there is no royalty fee for redistributing the Intel IPP libraries with your software. By licensing the Intel IPP product for your developers, you have redistribution rights to distribute the Intel IPP library files with your software for an unlimited number of copies. For more information please refer to the end user license agreement.
- ***How many copies of the Intel IPP product do I need to secure for my project team or company?***
  The number of copies of the Intel IPP product needed is determined by the number of developers who are writing code compiling and testing using the Intel IPP API as well as the number of build machines involved in compiling and linking, thereby needing the full development tools file set of Intel IPP product.