

## Porting Phission to the Blackfin/VDK

The project to port Phission to the Blackfin/VDK consisted of a few steps. First, the Phission source and header files had to be brought into the VisualDSP++ environment. Certain header and source files were not included in the new VisualDSP++ project as they were system dependent and the Blackfin/VisualDSP++ environment clearly wouldn't support them (i.e. SDL, X11, GDI, etc.) After the creation of the VisualDSP++ project a series of steps was taken to compile Phission. Anything that didn't compile was marked and commented out of the code so it could be resolved correctly later on. This changed the current setup of conditional compilation. The usual scenario was to compile for either a Linux system and compile for a Win32 system were Linux not the current system. Win32 compilation is now decided based on the existence of the WIN32 definition instead of the lack of a Linux definition.

After compilation issues were resolved with the VisualDSP++ project, the result was static `phission.h` and `phissionconfig.h` files that define what header files are part of the project and the settings of the code base. The `phission.h` file contains includes (in a particular order) of all the Phission header files so that user code only includes one file. The `phissionconfig.h` file tells the phission code what to headers can be included and what features are available. For example, whether JPEG compression is available or whether the `stdio.h` file is available. In addition, the `phStdint.h` files was created to define all the values that `stdint.h` would define were it included in the VisualDSP++ header files.

The second step was to attempt to code support for the operating system constructs that provide the portable execution layer of Phission. This could only take place once all the Phission compilation issues were resolved. The operating system classes include: `phMutex`, `phSemaphore`, `phCondition`, `phRWLock`, `phThread`, `phTimeStamp`, `phSocket` and `phServerSocket`. The `phCondition` and `phRWLock` features required the use of an algorithm that also is used for Win32 systems. These algorithms simulate the proper (or very close to) behaviour of their respective constructs by using semaphores, mutexes, private variable settings, queues and (for the `phRWLock`) a couple condition variables. The `phSemaphore` class was a straight forward wrapping of the VDK's (VisualDSP++ Kernel) API for semaphores. The method for using the time stamps was changed to be based on the `clock.h` implementation.

This consists of the `clock()` function and the use of the `CLOCKS_PER_SECOND` compiler definition. This remains to be changed since the `CLOCKS_PER_SECOND` compiler definition isn't good for a library project because an application could define it to be something different and the library (having been pre-compiled) will not see this change. The proposed solution will use VDK API calls to get a time stamp and the ticker per second information returned by a different API call.

The most difficult issue with the port was the creation of the `phMutex` class as the VisualDSP++ environment didn't have any released API for mutexes. The VDK's semaphore class had to be used as a binary semaphore to provide the mutual exclusion. This quickly turned out to be the wrong solution. Many Phission classes have `phMutex` classes protecting their changing of internal private member variables. There is a maximum limit of 508 user semaphores and this number was quickly overrun. It wasn't very straight forward since no documentation says that this is the absolute limit on how many semaphores a system can have. The immediate solution to this issue was to create a spin-lock that used an unscheduled region (to prevent other threads from presenting race conditions), an integer variable (to specify whether someone owned the lock), a loop on that variable and a call to yield the processor

to another thread ( to prevent hogging the processor ). The final answer to this problem was contacting Analog Devices product support, at which point they provided an unreleased API to use an internally implemented recursive mutex (RMutex) which resulted in a straight wrapping of the mutex API.

Once all the support for the VisualDSP++ Kernel was coded, work on the socket API provided by the LwIP ethernet facilities took place. All the code required the use of example code. Since Phission is meant for redistribution, as source and as a library in compiled form, the VisualDSP++ project is a library project. The library project needs to easily produce a single directory of header files and a single directory with the library file. To create the header file “include/” directory, one of the Automation engine example scripts was modified to take every “.h” file that was part of the Phission project and copy it to that directory. The library directory “lib/” must already exist for the distributable library output and a Post-Build command was added to copy the .dlb output to the “lib/” directory. This was absolutely necessary as there can be a Debug or Release build and the example code must use whatever the current build is to prevent using old code.

The sockets API for the LwIP distribution doesn't support ARP lookup tables, which resolve a hostname to an IP address. For this reason, certain hostname resolution code needed to be removed and replaced with the `inet_ntop()` function call. Again, another unsupported function that needed to be coded to support translation of an IP address to a string. There were also some modification required to support compilation with the LwIP headers that were the result of linker issues.

Once all the support for network and operating system constructs was complete, it was time to add features. Since the debugging of the system was going to be implemented by sending images over the network, JPEG compression was going to be used to reduce the transmission overhead. An IJG JPEG library project was created to compile the standard JPEG compression library for the Blackfin system. This was very easily accomplished since it's the benchmark library and completely written in C. A similar approach to the library redistribution was taken for the JPEG library as was done for the Phission library.

The first major example written to test the Phission code was the `phThread` test. The examples are written to use VDK boot threads that call the actual example code. Since Phission is meant to use the network functionality, there are two boot threads. The first is a modified version of the `lwip_sysboot_threadtype` that will post a semaphore to tell the second thread type (`PhissionBootTestThread`) when the network connection has been established. Once the semaphore has been posted, the `PhissionBootTestThread` will do whatever other setup is necessary. Currently there is no setup code required but this could potentially be any other type of a thread that manages some queuing mechanism that might handle a memcopy simulation that is implemented using Memory-to-Memory DMA. After setup, the example's main function is called and the test code proceeds to execute. The `phThread` test ended up raising a few issues of stack size, priority and process yielding. The stack sizes that are default for Phission are ~1500Kbytes, priorities default to 10 and the process yielding calls the `Sleep(1)` call instead of the appropriate `yield` call. This needs to be changed by resolving the yield issues and is planned for a later time.

The Second example written was to test out the blobbing code. This example takes two images files that were created with the GIMP software. The images files are saved as C code (a very useful feature of the GIMP) and compiled into the program. The images consist of a set of red squares that the image code can segment, thus exercising the blob algorithm code on the Blackfin. However, the output of the blob algorithm code is a list of blobs in the contained image that are sorted using the quicksort algorithm. The standard issue quicksort uses recursion to sort and so the stack limits were quickly overrun. This prompted the recursive implementation to be replaced with an iterative approach using a “stack” structure (linked list with First In Last Out properties).

The final example was a program in which to develop the OmniVision camera capture code. Before this could be done, code was tested in a standalone (sans Phission) application. The application required the modification of code released to our lab that worked for an OmniVision 7648 camera

module through the PPI port. This code makes use of the system services library and EZ-Kit utilities. The 7468 camera driver was converted to the 7620 module that is planned for distribution with the HandyBoard system. It was mostly a simple case to figure out the structure of the driver and its error checking facilities and then converting the register set structure to match that of the 7620. All TWI code and PPI code is done using the System Services Library implementation where the application demo given was using a custom/older version for the standalone application.

The demo application was run using the Image Viewer to monitor what the code was acquiring from the camera module. First YUV was being used as the format from the camera module and some issues were run into with that format. The actual information is received as a YUYV, UYVY, YVYU, etc. type byte array. Certain camera register settings determine the actual output and finally a format that the ImageViewer could interpret correctly resulted. However, this required translation as each 4 bytes represents two pixels where the U and V are duplicated. The same was later found out for the RGBG format that every 4 bytes represents a macro pixel.

The demo application code was re-written into a different structure of functions and then ported into a Phission capture class. The open and close calls performed all the setup and cleanup of the Blackfin system services that were being used. A problem arose when the standalone application's code to initialize the interrupts of the system was done. Since the ethernet drivers had previously performed this operation, the information contained within a global data structure was over-written by the OmniVision capture class 'open' code. It wasn't straight forward to debug this and required a week or two of stepping through code at the assembly level and source levels. Eventually this led to the answer that the interrupt manager information was incorrect and caused a jump to an invalid location. The final step was to figure out that the initialization code in the OmniVision capture module overwrote this and then the project was able to continue on with development.

Finally, an image was being captured and sent out of the EZ-Kit over a wireless bridge connection to a Windows laptop. This image was JPEG compressed and all operations were being performed within the Phission system. The main code was very short and simply linked an `phOmniVisionSource` class output to the `NetDisplay` input. The `NetDisplay` then displays for a `phNetSource` class that runs on the Windows laptop in a simple network viewer example (`phNetSourceTest`).

One of the first issues encountered were that the YUV-ish camera module output has to be translated into RGB to be handled by Phission and this required a significant CPU power to do the floating point conversion. This could be changed in the future by supporting the packed YUV format which would make the conversion a simple case of reordering bytes in the same manner as how the RGBG format is handled. The RGBG format was the answer to reduce the CPU overhead for floating point conversions. The next issue is still unresolved and is the misalignment of the image within the image buffer being captured from the OV7620 camera module.

Future progress on this project will be accomplished to resolve the misalignment issue with the image and make speed up some other operations. The byte reordering and repacking can be done completely with DMA movements by chaining DMA operations together. Additionally, there must be some JPEG compression code optimized for the Blackfin processor that could be used instead of the IJG distribution. Cleaning up the time stamp code to use code that can be compiled and distributed without state conflicts of the true system tick period will also be done. Fixing the yield scheduling issue that was solved with the `Sleep(1)` call must absolutely be done. Performing all 'memcpy's using a Memory-to-Memory DMA call will undoubtedly speed up the system as long as it is even possible to simulate memcpy using that facility. Finally, adding a packed YUV format to the Phission supported formats will be useful for matching both possible formats that can be captured from the OV7620 camera module.