Memory Usage of a Software Communication Architecture Waveform

Philip J. Balister
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
balister@vt.edu

Carl Dietrich
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
cdietric@vt.edu

Jeffrey H. Reed
Wireless@VT
432 Durham Hall, Mail Code 350
Blacksburg, VA 24061
reedjh@vt.edu

***Abstract*— One question commonly asked about software defined radios is "How much memory do I need?". While the answer depends greatly on the software framework and underlying operating system, this paper describes tools used to measure memory usage on a Linux based system running the OSSIE SCA framework. In addition to the conventional tools commonly used to study memory usage, this paper introduces the *exmap* and *exmap-console* tools for performing detailed memory usage measurements. Results from these measurements help define how much memory a radio requires by accurately measuring memory usage. Accurate memory usage measurements allow the system designer to reduce the number of memory chips in the final hardware design, this leads to lower cost radios that require less power to operate.**

I. Introduction

A Software Defined Radio (SDR) is a radio whose function is defined by software, not by the design of the underlying hardware. SDR technology provides radio users with much greater flexibility than is available from a traditional hardware defined radio. When a new standard is developed, rather than replacing the entire radio, only the software needs replacing. When two groups of people, who normally use incompatible standards must work together, their radios could be loaded with software that allows them to communicate with each other. Over the lifetime of a hardware design, new improved radio standards can be installed on the radio without requiring new hardware.

These characteristics of SDR provide cost savings by extending the life cycle of hardware and provide more capability from a given set of hardware.

For small form factor radios such as hand held radios, or sensor radios, SDR designs provide many benefits, however, a SDR design will use more power to operate than traditional fixed hardware solutions. This is due to the power consumption of data converters and signal processing hardware. Close attention must be paid to the SDR hardware in order to meet battery lifetime requirements. Some factors that impact power consumption are total memory requirements, system clock rates, data converter sample rates. This paper focuses on techniques to analyze system memory usage.

II. Embedded Systems

An embedded system is a microcomputer system with application specific hardware. Rather than the standard peripherals found on a personal computer (PC), an embedded system contains application specific peripherals. Many embedded systems are designed for battery powered operation, although some embedded systems, such as automobile engine computers, building climate control computers, and security system computers do not have low power requirements. On the other hand, embedded systems such as personal digital assistants, cell phones, and sensor network controllers, have stringent battery life requirements.

A typical SDR embedded system contains familiar peripherals such as sound interfaces, network interfaces, LCD panels and keypads. There are also specialized peripherals that provide interfaces to the RF circuitry, such as tuners, data converters, FPGA's and DSP's. A tuner converts RF signals from the antenna to the frequency needed by the data converter. Analog and/or digital tuning systems convert fixed, or variable frequency ranges to frequencies usable by the data converter's. The data converters convert analog signals to streams of digital numbers (or vice versa). These data streams are digital representations of the analog signals. FPGA's and DSP's perform high data rate signal processing on the data streaming from the data converters. The typical embedded system general purpose processor (GPP) cannot process data at the data rates used by the data converters.

III. Software Communications Architecture

The JTRS [1] [2] program is developing radio systems using SDR technology for military applications. The JTRS program seeks to develop a SDR capable of voice and data operations to replace a large collection of legacy radios used by the American military. Furthermore, the US military is transforming the way it operates, part of this effort is the development of the Global Information Grid (GIG). The GIG seeks to network the individual war-fighter to the command centers.

Part of this effort was the development of the software communication architecture (SCA). The introduction to the SCA specification [3] states that;

The SCA has been structured to:

1) *provide for portability of applications software between different SCA imp lementations,*

2) *leverage commercial standards to reduce development cost,*
3) *reduce software development time through the ability to reuse design modules,*
4) *build on evolving commercial frameworks and architectures.*

In order to meet these goals, the SCA defined a component based framework for implementing the functions required for a SDR, defined an operating environment for the software, and uses well defined standards for inter-component communication and configuration data storage. [4] provides a detailed history of the origins and evolution of the SCA.

Component based development is a key feature of the SCA architecture. Component based development has several basic concepts: each portion of the software must use well defined interfaces, components communicate with each other using standard communication packages, and components are developed with a standard operating environment. This provides a structure that allows easy re-use of components across different radio platforms. UML (Unified Markup Language) [5] is a standard graphical standard for describing component based software systems.

The operating environment for the SCA is based on several industry standards. The SCA specification describes the operating environment in great detail. Summarizing the specifications: The operating environment defines a subset of the POSIX interfaces for use by components, and the components may use the interfaces defined by minimum CORBA, and the IDL interfaces provided by the CORBA Naming Service and the CORBA Event Service. For file IO, the SCA provides file I/O services and requires the components use those interfaces, not other file IO interfaces.

The SCA uses CORBA[6] for inter-component communication. CORBA is a middle-ware standard developed by the Object Management Group (OMG). Numerous CORBA implementations are available, both commercial and open source. CORBA is used for communication services in a large range of application, such as banking, control systems, and many other applications requiring distributed computing in a heterogeneous computing environment.

A. OSSIE

OSSIE [7] (Open Source SCA Implementation::Embedded) was released by Virginia Tech in 2004. OSSIE is an open source SDR software framework based on the SCA. OSSIE originally used TAO for CORBA support and XERCES-C for XML support, later versions use omniORB for CORBA support and tinyXML for XML support. The first version of OSSIE ran on windows, subsequent versions run on Linux.

OSSIE was originally to introduce communication engineering students to SDR design methods, and provide a framework to support SDR research [8].

IV. Memory Management

The Memory Management Unit (MMU) controls a processes access to physical memory. The MMU provides the ability to allow multiple processes to use the same virtual memory addresses, while providing isolation between the processes. This simplifies software design since the programmer does not need to account for the details of memory addressing. The MMU maps the virtual address to specific physical memory addresses. Furthermore, the MMU allows programs to share the same physical memory under certain circumstances.

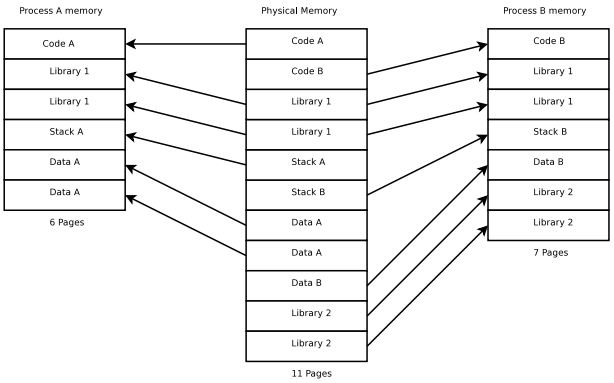

Fig. 1. Virtual Memory Management

A an example of how a MMU works is shown in Figure 1. This figure shows two processes (A and B) accessing the same physical memory space via a MMU. The physical memory is mapped into each processes by the MMU. Memory is mapped into the virtual memory space in units called pages, a typical page contains 4096 bytes of physical memory. The figure shows examples of pages used only by a process and pages shared between the two processes. The two processes each have unique pages and shared pages. Here are some terms used to describe MMU operation:

- Page - Smallest unit of memory the MMU can control.
- Anonymous mapping - Memory that does not have any relation to a file. Typically read/write memory for a process without any initialization.
- Dirty Page - A page that contains modifications.
- Copy on Write (COW) - A page may be shared between two processes until one of them writes new data into the page.
- Page fault - occurs when the processor attempts to access virtual that does have corresponding physical memory.
- Swap - writing dirty pages to mass storage temporarily to free physical memory for re-use.

V. Measurement tools

Measuring physically memory usage in virtual memory systems can be challenging. With conventional command line tools, such as *ps*, *top*, *pmap*, and *free*, estimates of memory usage can be created. However, results obtained from these tools do not fully account for the impact of memory shared between processes. The *exmap* [9] tool provides detailed measurements of process memory usage for desktop class systems. For embedded systems, the *exmap-console* [10] extension provides a small server to install on the embedded machine allowing

```
             total  used  free shared buffers cached
Mem:         29676 15300 14376      0       0  10368
-/+ buffers/cache:  4932 24744
Swap:            0     0     0
```

Fig. 2. Output from *free* command

memory usage data collection on the embedded system and analysis on a desktop system.

The *free* command provides a system level view of memory usage. This command shows the total memory available to the system and with limited details of how the memory is used.

Figure 2 shows the output from the *free* command run on an OMAP Starter Kit[11] (OSK) shortly after booting. This board has 32 M bytes of RAM. At first glance the 15.3 M bytes of memory used seems high for an embedded system with only a few processes running, however the second line shows that, after accounting for memory used for buffers and caching, the system is using 4.9 M bytes. Also note that the output of the *free* command shows the system does not have any swap space available.

Buffer and cache pages are used to store copies of data from mass storage devices in memory temporarily, in case the data needs to be reused. This reduces access time to data stored on mass storage devices at the expense of memory usage. However, if applications need more memory, the memory used by the cache is reclaimed for application use. This is why the *free* command shows the overall memory usage, and the memory usage adjusted by buffer/cache usage.

The *ps* and *top* command show several measures of memory usage on a per process basis. These are the amount of memory in use (resident), the amount of virtual memory used (virtual), the amount of memory used for code (code), the amount of memory used for data (data) and the amount of memory that could be shared with other processes (shared).

The *pmap* command breaks down virtual memory usage by section. A section is a piece of memory that is initialized with data read from a mass storage device, memory initialized to all zeros, and uninitialized memory. Sections without corresponding data on a mass storage device are known as anonymous sections. *pmap* shows the access flags set by the MMU, a section may be readable (r), writable (w) and/or executable (x). Figure 3 shows results from running *pmap* on the process that starts the Domain Manager and Device Manager classes for OSSIE.

Pages that are not writable (r–, r-x) may be shared between processes, since the data contained on the page can not be changed. Read/write pages (rw-) may be shared, until one of the process sharing the page writes to it, when a write occurs the page is copied to a new page before the update occurs. This is an example of COW.

Each of the commands *free*, *ps*, *top*, and *pmap* provide pieces of information needed to analyze system memory usage. *free* provides a high level view of memory usage. *ps* and *top* provide per process usage. *pmap* provides details of the internal process space memory map. However, these commands provide a fragmented view of true memory usage.

```
684:   nodeBooter
00008000   12K r-x /usr/bin/nodeBooter
00012000    4K rw- /usr/bin/nodeBooter
00013000  376K rwx   [ anon ]
4003a000 1768K r-x /usr/lib/libomniDynamic4.so.0.7
401f4000   32K --- /usr/lib/libomniDynamic4.so.0.7
401fc000  200K rw- /usr/lib/libomniDynamic4.so.0.7
4022e000    4K rw-   [ anon ]
4022f000   16K r-x /usr/lib/libomnithread.so.3.2
40233000   32K --- /usr/lib/libomnithread.so.3.2
4023b000    4K rw- /usr/lib/libomnithread.so.3.2
4023c000 1340K r-x /usr/lib/libomniORB4.so.0.7
4038b000   32K --- /usr/lib/libomniORB4.so.0.7
40393000   44K rw- /usr/lib/libomniORB4.so.0.7
4039e000    4K rw-   [ anon ]
4039f000  788K r-x /usr/lib/libossieidl.so.0.0.4
40464000   32K --- /usr/lib/libossieidl.so.0.0.4
4046c000   72K rw- /usr/lib/libossieidl.so.0.0.4
4047e000  276K r-x /usr/lib/libossieparser.so.0.0.4
404c3000   28K --- /usr/lib/libossieparser.so.0.0.4
404ca000    8K rw- /usr/lib/libossieparser.so.0.0.4
404cc000  396K r-x /usr/lib/libossiecf.so.0.0.4
4052f000   28K --- /usr/lib/libossiecf.so.0.0.4
40536000   60K rw- /usr/lib/libossiecf.so.0.0.4
40545000  756K r-x /usr/lib/libstdc++.so.6.0.8
40602000   28K --- /usr/lib/libstdc++.so.6.0.8
40609000    8K r-- /usr/lib/libstdc++.so.6.0.8
4060b000   12K rw- /usr/lib/libstdc++.so.6.0.8
4060e000   24K rw-   [ anon ]
407ea000 8188K rwx-  [ anon ]
 total  41408K
```

Fig. 3. Abbreviated results from *pmap* command

The amount of virtual memory a process uses does not reflect actual memory usage. For example, when the omnithread library (part of omniOrb) creates a thread, a stack section with a size of 8188 K bytes is created. However, since only a few hundred K bytes are actually used on the stack, most of that memory is never mapped to physical memory.

The shared memory field available in *ps* and *top* shows only the amount of memory that could be shared. There is no information detailing if the memory is actually used by more than one process, and if so, how many processes use it. It is possible to create estimates of how much memory is shared by examining the output from *pmap* to identify sections of code that could be shared and counting the number of processes that use these sections. This gives a count of the number of processes that share the same libraries, however this does not show the processes actually share specific pages.

With the standard tools available with operating system, system memory usage can be examined in some detail. However, results from these methods involve some "hand-waving" to explain results obtained from these tools. A tool to accurately examine process memory usage would help reduce the amount of "hand-waving" involved in determining system memory usage. Furthermore, accurate measurements of memory usage help developers create systems that can operate with less physical memory, this can lead to cost and power savings in finished products.

A. Exmap and Exmap Console

Exmap is a tool that uses a Linux kernel module to access the page tables from the MMU. Examining the page tables

```
Process 1279 [nodeBooter]
 Virtual memory  :     24840 KB
 Effective VM    :     22285 KB
 Heap            :       264 KB
 vdso            :         0  B
 Mapped          :      5076 KB
 Effective mapped:      2521 KB
 Sole use        :       936 KB
```

Fig. 4. Output from *exmap* command

provides a wealth of information about actual memory usage. This information, combined with debugging information embedded in executable provides a very detailed view of system memory usage. *exmap* is designed for use on desktop class machines and is not suitable for use on embedded systems. *exmap-console* is an extension designed for limited resource systems.

In addition to the kernel module for reading the page tables, *exmap-console* contains three parts; the *exmap* program, *exmap-server*, and *exmapd*. *exmap* is the user interface program that reads data from the kernel module or a remote machine. The user interface provides information about memory usage, the amount of detail is controlled by command line options and the availability of the symbol table for the binaries. The *exmap-server* is a very small program that runs on the embedded system and allows the *exmap* user interface to connect to the embedded system via a network connection. This minimizes the system resources needed to monitor the embedded system. Finally, *exmapd* runs *exmap* periodically and logs results to disk. This allows monitoring of memory usage over a period of time.

For this paper, summary data was collected for each process using *exmap* via the *exmap-server* running on the embedded system. Data was also collected using the *free* command for comparison purposes. Only per process summary information is discussed in this paper, consult the *exmap* documentation for information on collecting detailed memory usage data.

Figure 4 shows the output from exmap for a specific process. At the time this snapshot was taken, the nodebooter process required 24.8 Mbytes of virtual memory. The effective virtual memory is the virtual memory number modified to account for actual shared memory. This effective memory calculation is described later in this paper.

The mapped amount of memory is 5.1 Mbytes of RAM, this is the actual amount of memory in use at this time by the nodebooter. The effective mapped memory is 2.5 Mbytes of memory. The sole use number is 936 kBytes of memory, this is the amount of memory used only by this process.

Effective memory is the sum of the sole use memory and the amount of shared memory mapped divided by the number of processes sharing that memory. By summing the effective mapped memory numbers for all process on a system, the total amount of memory in use can be calculated. For this paper, only the sum of the effective mapped memory for radio related components is used in calculations, this introduces small errors in the memory usage calculations due to memory shared with processes not directly related to the radio system.

Process	Mapped	Effective Mapped	Sole Use
omniNames	2292	1847	1800

TABLE I
MEMORY STATE AFTER STARTING CORBA NAMING SERVICE

VI. DATA RADIO SYSTEM SIMULATION WAVEFORM

In order to evaluate the embedded system resource usage, a waveform was developed using the OSSIE framework. The test system consists of the OSK and the waveform, the combination of the hardware and software provides a representative example of a SDR for measuring the system memory usage.

The waveform developed for the embedded system is a simple simulation of digital modulation transmitter and receiver. This is a simple waveform constructed from several. The components used are; symbol generator, channel simulator, and receiver. The operations in each of these components is greatly simplified from the operations performed in an actual radio. The waveform performs little signal processing, however, it provides a good skeleton for measuring the resource usage by the SCA portions of the system.

The waveform does three things; create packets of data representing a QPSK signal, offsets the constellation by a complex imaginary number, and prints the values of the received symbols. While this is a very simple waveform, the methods described in this paper can be applied to larger more complex systems.

VII. MEMORY USAGE EASUREMENTS DURING WAVEFORM START UP

Start up of a SCA waveform involves several distinct steps. During the startup process, measurements of system memory usage were made at easch step. This process provides insight into the memory usage for a SCA based waveform running on an operating system that supports shared memory. Measurements of each processes memory usage were made at the following points; CORBA naming service start up, nodeBooter start up, c_wavLoader start up, waveform installation, and waveform start up. By summing the effective mapped memory data, the amount of memory used by the radio software is calculated. Subtracting the sum of the sole use memory shows the amount of memory shared by the radio processes.

Table I shows the memory used by the CORBA naming service. Some sharing of memory already occurs due to the presence of system wide libraries, such as the standard C and C++ libraries. At this point, the radio software requires 1.8 M bytes of RAM. The 492 K bytes difference between the mapped and sole use memory is memory shared with operating system libraries, such as the C and C++ libraries. The memory shared amongst other system libraries introduces small errors into the calculation described above, since this memory is not accounted for.

Starting the nodeBooter process create instances of the Domain Manager and Device Manager SCA classes, and starts

Process	Mapped	Effective Mapped	Sole Use
omniNames	2612	1051	428
nodeBooter	5076	2521	936
GPP	4772	2285	760

TABLE II

MEMORY STATE AFTER STARTING NODEBOOTER

Process	Mapped	Effective Mapped	Sole Use
omniNames	2612	901	404
nodeBooter	5160	2111	940
GPP	4832	1866	760
c_wavLoader	4668	1775	716

TABLE III

MEMORY STATE AFTER STARTING C_WAVLOADER

the GPP device that provides the executable device that runs the waveform components. Table II shows memory usage by the radio after starting the nodeBooter; at this point, the radio software uses a total of 5.8 M bytes of memory. However, only 2.1 M bytes is used exclusively by the the radio software, the remaining 3.7 M bytes is shared by the radio processes.

c_wavLoader provides the user interface to the radio. c_wavloader calls the intrefaces provided by the Domain Manager in order to obtain radio information and status. Now the radio uses 2.8 M bytes plus 3.8 M bytes shared by the the radio software for a total 6.6 M bytes of memory.

Next the test waveform is installed on the radio. Waveform installation creates processes for each of the radio components by using the interfaces provided by the GPP device. Table IV shows memory usage after the waveform is installed. At this point the radio processes are using a total of 9.2 M bytes of memory, 5.4 M bytes are used exclusively by different radios processes and they share 3.8 M bytes of memory.

Finally, when the waveform starts, the memory usage numbers show no change. Table V contains these results. This is due to the waveform signal processing being very small compared with the framework code required to create the waveform. In a real waveform, we would expect memory usage to increase as the actual radio code started execution.

Table VI presents a summary of system memory usage during the start up process for the SCA waveform described in this paper. This table summarizes the results described above and adds a final column with an estimate of memory usage

Process	Mapped	Effective Mapped	Sole Use
omniNames	2692	758	488
nodeBooter	5372	1638	1036
GPP	4964	1288	712
c_wavLoader	4824	1305	776
TxDemo	5304	1441	792
ChannelDemo	5296	1438	792
RxDemo	5256	1421	784

TABLE IV

MEMORY STATE AFTER INSTALLING THE WAVEFORM

Process	Mapped	Effective Mapped	Sole Use
omniNames	2692	758	488
nodeBooter	5372	1638	1036
GPP	4964	1288	712
c_wavLoader	4824	1305	776
TxDemo	5304	1441	792
ChannelDemo	5296	1438	792
RxDemo	5256	1421	784

TABLE V

MEMORY STATE AFTER STARTING THE WAVEFORM

Step	Sole use total	Shared total	Total	From *free*
Naming service	1.8	.047	1.8	1.8
NodeBooter	2.1	3.7	5.8	4.3
c_wavLoader	2.8	3.8	6.6	6.2
waveform install	5.4	3.8	9.2	10.1
waveform start	5.4	3.8	9.2	10.5

TABLE VI

MEMORY USAGE DURING WAVEFORM START UP

based of results from running the *free* command at the points during the start up process as the *exmap* commands. These show that the numbers from *free* roughly follow the results from *free*, but the *exmap* results provide more details about the overall memory usage.

VIII. SUMMARY

This paper describes tools and methods used to measure system memory usage for a SCA based waveform running on a Linux based embedded computer. The basic memory usage tools (*free*, *top*, *ps*, and *pmap* provided with most Linux systems were reviewed. The *exmap* and *exmap-console* tools for performing detailed memory usage analysis for Linux based system provide several advantages over the traditional tools. Measurements of a simple SCA based waveform running on an OSK were collected and summary results from these measurements were compared with measurements made with the *free* command.

IX. ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant No. 0520418.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The authors also acknowledge support from the National Institute of Justice and Wireless@VT partners.

REFERENCES

[1] http://enterprise.spawar.navy.mil/body.cfm?type=c&category=27&subcat=60.

[2] "Joint tactical radio system - wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/JTRS.

[3] "Software Communications Architecture Specification," Final/15 May 2006 Version 2.2.2, Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS), May 2006. Also available at http://jtrs.spawar.navy.mil/sca/.

[4] J. Bard and V. Kovarik, *Software Defined Radio: The Software Communications Architecture*. Wiley, 2007.

[5] M. J. Chonoles and J. A. Schardt, *UML 2 for Dummies*. Wiley Publishing, 2003.
[6] "Welcome To The OMG's CORBA Website." http://www.corba.org/.
[7] "Ossie - trac." http://ossie.wireless.vt.edu/trac.
[8] Max Robert, Shereef Sayed, Carlos Aguayo, Rekha Menon, Karthik Channak, Chris Vander Valk, Craig Neely, Tom Tsou, Jay Mandeville and Jeffrey H. Reed, "OSSIE: Open Source SCA for Researchers," SDR Forum Techical Conference, 2004.
[9] "Exmap." http://www.berthels.co.uk/exmap/.
[10] "Exmap console." http://projects.o-hand.com/exmap-console.
[11] "Spectrum Digital Inc. OMAP5912 Starter Kit (OSK)." http://www.spectrumdigital.com/product_info.php?cPath=31_80&products_id=39&osCsid=eb4c3f4c8acf067d1ccac49a6436c43d.