21520 30th Drive SE #110
Bothell, WA 98021 USA
Tel: (425) 951-8086
Fax: (425) 951-8095
sales@datalight.com
support@datalight.com
www.datalight.com

Memory implications for SOCKETS™

WHITEPAPER SERIES

Overview

Working memory is a very scarce resource in the SOCKETS system. Since SOCKETS is deployed in the embedded systems environment, it is important to keep the memory footprint as low as possible. For that reason the working memory is restricted to a maximum of 64 kilobytes and often less than that is allocated. This allocation takes place at load-time and can be specified by the /m and /s command line options. The working memory is used for internal variables, the program stack and the memory heap. Memory is dynamically allocated and de-allocated from the memory heap for structures like interface control blocks and TCP connection blocks as well as for data buffers for incoming and outgoing data at various levels.

In order to reduce the total footprint of SOCKETS, program code only used for initialization is stored in the data segment and released for dynamic use when initialization has been completed. It is thus possible for initialization to fail because of insufficient memory, but to succeed when memory is increased slightly and still have sufficient memory available for SOCKETS to operate satisfactory. The print services of SOCKETS are structured in a similar way; if not configured at all, the code memory will be utilized as dynamic memory.

The subject of dynamic memory requirements is quite complex since it depends on many factors which can be difficult to quantify including the nature of the traffic, the configuration and type of peer, the nature of the application programs and the performance/memory use trade-offs the system can tolerate. This whitepaper will attempt to quantify some of the memory requirements and point out performance/memory use trade-offs which can be configured.

General considerations

When SOCKETS runs out of stack memory, it will always lead to a fatal error condition (crash, freeze, bomb-out, hang or whatever terminology you care to use, including "stop responding"). It is therefore important to monitor the stack usage at least at development time using MEMSTAT.EXE or the appropriate API calls and ensure that the minimum stack size observed should always be a few hundred bytes.

When SOCKETS runs out of heap memory, it is not so bad since it will normally only lead to a performance degradation. When it happens too often, though, the performance can be downgraded to none at all, which is not much different from the fatal error condition discussed above. The ideal condition is that MEMSTAT.EXE or the API calls should never show a non-zero memory allocation error count.

The memory management used by SOCKETS will always allocate and free dynamic memory in chunks of 4 bytes with an overhead of 4 bytes. To allocate 1 byte requires 8 bytes of memory, to allocate 4 bytes also requires 8 bytes and to allocate 1000 bytes requires 1004 bytes. When a block of memory is allocated, two of the overhead bytes are used to specify the size of the block and two bytes to specify the address of the block. The latter is used as a "sanity check" when the memory block is freed. If the address had been corrupted, the entire block is discarded to prevent the entire free chain to be corrupted. MEMSTAT.EXE and the API calls will indicate this as "memory free errors". It could be the result of a hardware fault, corruption by another software module like a user application or a Packet Driver or an internal SOCKETS error. Once the possibilities mentioned first have been eliminated, report the problem to Datalight.

Some of the heap memory usage of SOCKETS is determined by the configuration, is allocated at initialization time and stays constant while SOCKETS is loaded. This is particularly true about interface control blocks, socket structures and static routes. Other usage is more dynamic and varies over time, like connection control blocks, ARP entries and dynamic (RIP) routes. Yet another class of memory usage is highly dynamic and difficult to predict and is mostly related to the use of memory buffers for sending and receiving data. This is especially true of memory requirements for TCP data transfer which is highly dependant on configuration values for TCP. Note that some configuration parameters are determined by the peer TCP which may not be a SOCKETS implementation and consequently not under our control. This can be even worse when using SOCKETS as an IP router carrying TCP traffic between two third party peers, in which case we may have **no** control over the TCP configuration parameters.

Sockets

The number of normal and DOS-compatible sockets, is related to fixed heap memory usage as follows:

Mem_sockets = 78 * normal_sockets + 4 bytes

Mem_DC_sockets = 78 * DC_sockets + 4 bytes

Since DOS-compatible sockets are only used for some third-party legacy applications, some 940 bytes of heap memory can almost always be saved by using the /d=0 command line switch when loading SOCKETS. Also ensure that no more than the required number of normal sockets is configured by the /n=<number> command line switch. Note that the default number of DOS-compatible sockets is 12 and that of normal sockets is 8. There is no need to consider the number of sockets for running SOCKETS applications like FTP, TNVT, HTTPD, FTPD or HTTPFTPD since they do not use the CAPI.

Interface

The memory usage requirements are different for the different interface types.

Packet Driver

When using one or more Packet Driver interfaces a fixed amount of memory is required:

Mem_PD_fixed = max_mtu + 132 bytes

max_mtu is the highest value of MTU used by all the Packet Driver interfaces. If only one is used, it is simply the MTU value of that interface.

For each Packet Driver interface:

Mem_PD = 220 bytes

When receiving data, 12 bytes plus the actual size of the Packet Driver packet is allocated. For DIX Ethernet that is 14 bytes plus the IP payload. It is slightly more for Token Ring. The IP header is normally 20 bytes and the TCP header another 20 bytes long. The overhead for TCP data is thus 66 bytes. Remembering that memory is allocated in 4 byte increments, the memory required for the minimum size TCP segment e.g. just an acknowledgement, is 68 bytes. Some Packet Drivers require a minimum of 64 bytes to be allocated, bringing the memory required for the smallest TCP segment to 72 bytes.

When using any high throughput device technology like Ethernet, there is a possibility of data being received at such a rate that it can't be handled immediately and must thus be queued. In order to prevent such a queue from consuming too much memory, the *numbuf* parameter in the *interface* command is used to limit the number of packets to buffer. If maximum size datagrams (size = MTU) are being received, the maximum memory required for Ethernet packets is *numbuf* * (MTU + 26) rounded up to the nearest 4 byte boundary. Having an MTU of 1500 with *numbuf* = 5, may require 7640 bytes per Packet Driver interface.

Data being sent via a Packet Driver is not queued by SOCKETS, but may be queued by the Packet Driver itself. It may thus require memory usage by the Packet Driver, but none by SOCKETS.

SLIP

For each SLIP interface:

Mem_Slip = 328 bytes

When using any serial interface, including SLIP, a number of datagrams may be queued if it supplied faster than it can be sent by the serial interface. This may lead to excessive memory usage especially if the interface is being flow controlled by a modem or a directly connected peer. The *buflim* parameter in the *interface* command is used to set a limit to the number of datagrams which will be queued before starting to discard them. Each datagram being queued has an overhead of 12 bytes. Let us take a worst-case scenario where we have specified an MTU of 1500 and a *buflim* of 30 and we are getting maximum length datagrams. We will require 1512 * 30 = 45360 bytes just to accommodate the queue. Note that this is a problem which may be encountered when handling routed datagrams of any type, ICMP or UDP data or TCP data from a number of simultaneous connections. A single TCP on a properly configured system will not run into this problem since the breakdown in the normal flow of acknowledgements will prevent it.

When receiving data a fixed buffer of MTU + 12 bytes is allocated for each packet which is equivalent to one datagram. This buffer is not truncated if less data is received and is passed through the protocol layers until it is either delivered to another interface in the case of IP routing or to the same interface in the case of an ICMP echo request converted into an ICMP echo response, delivered to an API or discarded for any reason, including having being used to update TCP status if it contained a TCP acknowledgement. When receiving a lot of small UDP datagrams without prompt acceptance by the API, memory problems may result. Consider the case where an MTU of 1500 is used and 10 UDP datagrams each containing one byte of data are received. Although only 10 bytes of user data are awaiting delivery, 15 120 bytes of memory are being used!

CSLIP

For each CSLIP interface:

Mem_cslip = 4424 bytes

The high memory requirement is a result of using Van Jacobson TCP/IP header compression.

The *buflim* issues discussed under SLIP, are also valid for CSLIP and so is the received data issues.

PPP

When using one ore more PPP connection a fixed 516 bytes is required.

For each PPP connection:

Mem_PPP_C = 4624 bytes (using Van Jacobson TCP/IP header compression)

Mem_PPP = 584 (not using Van Jacobson TCP/IP header compression)

The *buflim* issues discussed under SLIP, are also valid for PPP and the received data issue is even worse. For each received packet, which may contain a PPP control packet or an IP datagram, 2 * MTU + 24 bytes are allocated. Using the same example as before, we need 30 240 bytes to buffer the 10 bytes worth of user data.

Modem

Each modem configuration file requires approximately 60 bytes + the number of bytes in the modem commands. Comments do not require any memory.

ARP

Each ARP entry requires 56 bytes. An ARP entry is required for each host on a LAN with whom any communication has taken place in the preceding 15 minutes.

IP

Each route in the system requires 44 bytes. In a statically configured system only a small number of routes are used, but in a badly configured system using RIP, a large number of routes may be generated. In general one should be very careful when using both "*rip use"* and "*rip advertise"*. Use *"rip use"* only in non-routing hosts and *"rip advertise"* only in routing hosts.

UDP

Each UDP "connection" requires 24 bytes. If CAPI or BSD Sockets are used, each UDP socket will also take up one of the pre-configured sockets described under the heading **Sockets**. Sending and receiving UDP datagrams need additional data buffer memory which will be described under **Data Buffers**.

TCP

Each TCP connection requires 196 bytes. If CAPI or BSD Sockets are used, each TCP socket will also take up one of the pre-configured sockets described under the heading **Sockets**. Sending and receiving TCP data need additional data buffer memory which will be described under **Data Buffers**.

When using the SOCKETS system as the end point of a TCP connection, the TCP *mss* and *window* parameters play an important role in memory usage. Note that third party TCP traffic being routed by SOCKETS are not influenced by these parameters at all.

The TCP *mss* parameter determines the maximum size of a TCP segment which a peer will send and is negotiated at connection establishment time. SOCKETS uses *tcp mss* or *mtu* – 40 of the interface through which the connection is established, whichever is less, to set the actual MSS. The magic number of 40 is the size of a normal IP header (20 bytes) plus the size of a normal TCP header (20 bytes). The TCP *window* parameter determines the largest window which will be offered to the peer. It is recommended that *window* is an integer multiple of *mss*, with a value of 2 resulting in a good speed/memory usage compromise. For Ethernet *mss* is typically set at 1460 and *window* at 2920. For a peer sending maximum sized segments as would occur with an incoming FTP transfer, it means that acknowledgements in the reverse direction will always overlap with incoming data transfers while the maximum memory being used per connection to queue the incoming data will be 2 * (MTU + 28) = 3056 bytes for the typical Ethernet case. If we have very little memory or many simultaneous connections, a good tradeoff is to set *window* equal to *mss*. To save even further, one can reduce the value of *window* and *mss*. Keep in mind that there is a fixed overhead of 40 + 28 = 68 bytes. If we set *mss* to 68, it means that we will have a 50% overhead; which may even be acceptable. In the case of a serial interface, especially PPP, it makes even more sense to reduce the MTU. The overhead of 40 bytes only applies to SLIP. For CSLIP and PPP using TCP and header compression, the header is compressed to between 3 and 5 bytes. This will have a speed improvement and not a memory usage improvement.

Suppose we want to speed up the incoming transfer in an Ethernet system by setting *window* to 6 * *mss* = 8760. If we have used a *numbuf* value of 5, we will notice a dramatic **reduction** in performance. The reason is that the peer will attempt to fill up the window and send 6 segment in rapid succession. Unless we have a really fast CPU, it will result in the Packet Driver interface having to queue 6 packets, but the *numbuf* parameter will only allow 5, resulting in the packet and thus the IP datagram and thus the TCP segment to be discarded and the peer having to re-send it after a timeout period. This is not really a memory allocation problem, but memory allocation problems can present themselves in much the same way: an unexpected reduction in performance.

When sending TCP data we are at the mercy of the peer. SOCKETS will not allow more data to be queued than the size of the offered window minus the data in transit i.e. the data which has not yet been acknowledged. The *mss* offered by most systems will be 1460 for Ethernet systems; this is a result of the maximum data size of an Ethernet packet being 1500. Unix, Linux and Windows systems can offer a window size of the order of 60 000 bytes. If the user application does not regulate the sending of a constant stream of TCP data, it may well consume all of the available SOCKET heap memory. This will result in memory errors which **will** regulate the flow, but which may also have negative effects on other connections or even incoming data on the same connection. In such a case it is a good idea to determine the amount of queued data by using the **GetKernelInformation(**K_INF_TCP_CB**)** API function and to not send more than 2 * tcb.mss – tcb.sndcnt bytes at any time. Alternatively the NET_AS_XMT event can be used by utilizing the **SetAsyncNotification()** to only send data when an acknowledgement of earlier data has been received.

Data Buffers

When data is sent from an API function like **WriteSocket()** a buffer with the size of the data plus a 12 byte header making it possible to queue the data is allocated and the user data copied into it. This buffer will move through the TCP or UDP layer, the IP layer and the Interface layer without being copied, but with headers from the various layers being inserted in the front. In the case of TCP a number of the data buffers allocated by the API layer, may be queued together before the TCP, IP and Interface headers are pre-pended.

It is important to recognize that sending many small chunks of data from the Application layer using TCP, can be very inefficient as far as memory usage is concerned. Suppose the user application wants to send 1000 bytes, one byte at a time. Keeping in mind that memory is allocated 4 bytes at a time, 16 bytes is required for each user byte, requiring 16 000 bytes just for the data queue. If the user application instead made 2 API calls with 500 bytes each, only 1024 bytes is required.

The issues regarding the size of data buffers for receiving data have already been discussed under the Interface headings.

Conclusion

From the information presented, it is clear that the issue of memory usage is not a simple one. Even after careful analysis, it normally boils down to experimentally determining what will work. It does help, though, to understand the issues involved.