CBASIC Application Tools
Toshiba Global Commerce Solutions

9/10/2013

1.0 Purpose

The Application Change Team has developed some tools and Serviceability improvements in the CBASIC Runtimes that we expect will improve the serviceability and supportability of the CBASIC Applications on the 4690 Operating System (OS). The executables found at the FTP link associated with this document run on Windows™ (.EXE) and/or 4690 OS (.286), depending on the extension.

The function in the runtimes included here is being shipped with the 4690 OS on the Optional diskettes. As long as you have a relatively recent level of the OS (since January 2005), you have runtimes that include the most important enhancements..

2.0 CBASIC Runtimes Enhancements

There are two versions of the Runtimes; SB286L.L86, which runs on the Controller for tasks such as Menu-driven applications, Checkout Support and Background tasks, and SB286LT.L86, which is only used in the Terminal to support Terminal Sales activities. Serviceability improvements were added to both, however the specific set of serviceability improvements implemented for Controller applications does not exactly match the set of serviceability improvements for Terminal applications.

These changes were added over time, so the specific feature you want or need might not be in the version of the Runtimes you are currently using. See the tables on page 27 to determine the changes made for a specific version of the Runtimes.

2.1 Error Trapping – Terminal and Controller

In order to assist programmers in isolating coding errors that cannot be re-created with the debugger, it is a usual practice to force a dump when the unique error occurs. One way is to add a test in a user exit, and a call to `ADXSERVE` so that a dump occurs at that point. While this is not a 'bad' thing to do, it requires a source code change. Trapping the error in this manner also causes the stack to become 'reset' and it becomes difficult to determine where the problem occurred in the application. If testing for the trap can be moved into the runtimes, then the dump can be taken while the stack is intact and the dump becomes much more valuable.

The runtime modules (SB286LT.L86 and SB286L.L86) have benign, inactive code that can be activated by the `SETTRAP` program. *You do not normally have to change your source code or relink your program to use this type of trap!* When the specific runtime error occurs, and optionally a session number associated with that error, the application abends at that point and a dump is captured if the Application Dump flag was set to `Y`. The dump will have a stack that identifies the active call path and Code Address execution point where the failure occurred. The Application dump flag is set for Controller applications in "Controller Configuration" - "Controller Characteristics", or is set for Terminal applications in "Generic Terminal Configuration" - "Load Definitions" - "General Settings".

This Runtime trap only has value when the error that is occurring is fairly unique (it does not happen ordinarily). For example, you would *not* want to trap on an OE error unless it was for a specific session number. You also would not want to trap an OE on session 64, because session 64 is used for many different files.

See `SETTRAP` on page 25 for instructions on how to run that tool.

2.2 Call Tracing in Application Runtimes – Terminal and Controller

In the TGCS CBASIC Terminal applications, code was added many years ago to save the module call path into a string variable so that some determination can be made about the path through which the code flowed. However, this is rudimentary, only exists in Terminal applications, and has proven to be of limited use. A facility has been incorporated into the CBASIC runtimes that will save the address of each subroutine or function call in a trace buffer. GOSUBs are not saved. This happens automatically and does not require the application programmer to make any source code changes. The amount of code that was added to save these calls is relatively small, so performance degradation is usually unnoticeable.

Couple this tracing with the new version of the 4690 Application Debugger or Application Dump Analyzer (`APPDUMP`) and you will be able to see the actual call history that the application has performed. The new versions of the debugger and dump tool have been enhanced to display the name of the function or subprogram in indented fashion, depending upon how deep the call is. For example:

```
TSTPEC01
 FORMAT.AMOUNT
  BINPACK4
TSPREC01
```

The default number of trace entries is 250. Some problems require a much larger trace buffer to diagnose. See `SETTRACE` on page 24 for instructions on how increase the number of trace entries.

The SWI disable count and an `ON ERROR` counter are also saved in this trace. Both help to diagnose the source of incorrect handling in the application or its `ON ERROR` routines. Refer to Knowledgebase article "Why do programs abend with ERRN = 80004186?" for more information.

In TGCS Supermarket Application, IR50909 added code to the mainline EAMTS11C to call EAMTRACE as the *first* call. The runtimes will take note of the address of the first call and will never trace any calls to that function. The purpose of this is to reduce the noise of irrelevant EAMTRACE calls in this trace. If you do not have that APAR installed, you can add these two lines at the beginning of the mainline (EAMTS11C.BAS):

```
DIM TS.TRACE(25)    ! MODULE TRACE ARRAY
CALL EAMTRACE(5050H)! dummy call
```

In GSA, similar code can be added to the initialization so that the call to EMOD is done as the first call in the execution of the program.

Beginning with Runtimes version 67 (February 2011), the two-letter ERR code and the ERRF session number are unconditionally saved in the Call trace each time an error occurs. Two 16-bit integers are shown in hex for each of these trace entries. The first integer is the hex value of the ERRF session number that caused the error. The second integer is the two-letter ERR code, shown in hex. If both characters of the ERR code are * or A through Z, then APPDUMP also shows the ERR code as a two-character string. Here is an example. The error being traced is a CU error on session 50 (decimal).

```
185   14:09:21.158          2C 01   3    Runtimes trace: 0032 4355 "CU"
```

2.3 Timing in Milliseconds – Terminal and Controller

Most of the time, you can compare the time stamps of entries in the Application Call Trace to determine where performance issues are introduced. Some classes of timing problems however are difficult to solve without the use of a high-resolution timer and capturing of points in time using global variables that you interpret from a dump. The one-second granularity provided by TIME$ is often too imprecise to determine where time is being consumed in an application, because many events of concern can occur within a single second.

CBASIC Runtimes V59 provides a new function for Terminal applications to obtain an INTEGER*4 value of time. The returned value is the number of milliseconds past midnight for the current day. Use of this function provides much better ability to determine where time is being consumed in an application.

As an example, you might want to know the average time needed to sell an item. You can insert calls to `MILLISECONDTIMER` at strategic points in your code, subtract begin/end values, and then save the results in global variables or arrays so that you can view them in a dump or periodically write them to a file. Values can be saved as integers or can be converted to more useful forms such as “hh:mm:ss.xxx” for easier interpretation by a person. Here is an example of use:

```
INTEGER*4 GLOBAL MILLISEC  ! ms past midnight
STRING    GLOBAL MILLISEC$ ! hh:mm:ss.xxx
  ...

! This Runtimes function provides the time as
! the number of milliseconds past midnight.
FUNCTION MILLISECONDTIMER EXTERNAL ! in runtimes
INTEGER*4 MILLISECONDTIMER      ! returns INTEGER
END FUNCTION                    !

! This local function obtains the time in
! milliseconds past midnight, and converts
! it to a printable string of the form
! HH:MM:SS.mmm
FUNCTION MILLISECONDTIMER$    ! ms past midnight
STRING MILLISECONDTIMER$      ! returns STRING
INTEGER*4 HOURS, MINS, SECS, MS

MS = MILLISECONDTIMER ! call the Runtimes
HOURS = MS/3600000 : MS = MS - HOURS*3600000
MINS  = MS/60000   : MS = MS - MINS *60000
SECS  = MS/1000    : MS = MS - SECS *1000

MILLISECONDTIMER$ = RIGHT$("0"+STR$(HOURS),2) + \
              ":" + RIGHT$("0"+STR$(MINS ),2) + \
              ":" + RIGHT$("0"+STR$(SECS ),2) + \
              "." + RIGHT$("00"+STR$(MS  ),3)
END FUNCTION                                    !
  ...
MILLISEC = MILLISECONDTIMER   ! get integer time
MILLISEC$ = MILLISECONDTIMER$ ! get time string
```

Typically you would use only one of the two calls shown at the bottom of the code snippet above. The first gives an INTEGER*4 result. The second gives a printable string such as "10:32:54.356". If only the first form is desired, you do not need to code any of the function `MILLISECONDTIMER$`. In either case, `FUNCTION MILLISECONDTIMER EXTERNAL` must be declared.

2.4 Unresumed Error Trapping – Controller

This improvement attempts to identify functions or subroutines that allow an unresumed error to occur. It does so by saving the SWI Disable count at the beginning of each function or subroutine call and comparing it against the count when that particular function or subroutine returns. If the count is different, it *usually* means that an error was not resumed. An RN error is raised at that point.

This new capability is disabled by default, but can be enabled by running `SETTRAP MR` (missing resume). An RN error will be reported when a missing resume is detected. `SETTRAP RN MR` now does more than just set a trap for RN and MR, although the MR error code is never actually raised. The MR specification merely activates the dormant code in SB286LT.L86 to look for missing resumes. `SETTRAP` dated 02/20/2009 or later is required.

This new capability was used to quickly locate three unresumed errors in TGCS SA Checkout Support in February 2009 (IO10055).

2.5 Integer Tracing – Terminal

There are times when a specific application variable gets undesirably changed to a 'junk' value and you cannot determine why or where. This design change can help.

You now have the ability to automatically trace the values of your choice of up to three integers that have static scope (`GLOBAL,` or defined outside all functions and subroutines). The variables are traced every time a CBASIC function or subroutine is called, in similar fashion like an entry in the Application Call Trace.

Your application program can call a new, integer-trace subroutine to indicate what variable should be traced each time an entry is placed in the Application Call Trace. One, two or three such calls can be inserted. These calls do not explicitly trace the integer at that point in time; they specify the *address* of the integer that is to be traced *from that point forward*.

While `APPDUMP` is interpreting the Application Call Trace in a dump, it looks for these integer-trace entries, and interprets them on the same line associated with the Application Call Trace entry where they were saved.

For simplicity of implementation in the Runtimes, the number of variables to trace is limited to a maximum of 3, regardless of their size (1, 2 or 4 bytes each). All three can be I*1 or I*2 or I*4, or they can be any combination of sizes, and can be established in any order. Tracing of Strings or Real variables is <u>not</u> supported. Tracing of an Integer array or any of its elements is <u>not</u> supported.

To do Integer Tracing, you do not need to activate anything in the Runtimes, but you must make two types of minor source code changes:

- Declare the trace subroutine(s) you intend to call. Here is an example declaration for each of the three different subroutines.

```
! Trace 4-byte Integer
SUB TRACEINTEGER4(I4) EXTERNAL
INTEGER*4 I4
END SUB

! Trace 2-byte Integer
SUB TRACEINTEGER2(I2) EXTERNAL
INTEGER*2 I2
END SUB

! Trace 1-byte Integer
SUB TRACEINTEGER1(I1) EXTERNAL
INTEGER*1 I1
END SUB
```

- Call the appropriate `TRACEINTEGER`x subroutine *one* time for each of the variables you want to trace. You can use up to three `CALL`s. You determine the name of the subroutine to call based upon the size of the variable you want to trace. Here is an example that shows how they are called:

```
        INTEGER*4 GLOBAL GA.WAIT.TIME
        INTEGER*4 GLOBAL TS.NEG.FS.BAL
        INTEGER*2 GLOBAL TS.IN.IPL

        CALL DUMMY1

        CALL TRACEINTEGER4( GA.WAIT.TIME )
        CALL TRACEINTEGER4( TS.NEG.FS.BAL )
        CALL TRACEINTEGER2( TS.IN.IPL      )

        CALL DUMMY2

        GA.WAIT.TIME = 5
        TS.NEG.FS.BAL = 3

        CALL DUMMY3
```

For the example above, here is the `APPDUMP` output (with some columns of the Application Call Trace removed for brevity). Notice that in the `APPDUMP` trace below, variables `GA.WAIT.TIME` and `TS.NEG.FS.BAL` changed values between the `DUMMY2` and `DUMMY3` calls. This is confirmed by reviewing the source code above.

```
        There are 3 integer variables traced:
        051F:1C4E GA.WAIT.TIME
        051F:1C56 TS.NEG.FS.BAL
        051F:1B14 TS.IN.IPL

    SUB Application Call Trace
        (listed from oldest to newest):

                      ###   Time         Name
GA.WAIT.TIME
         TS.NEG.FS.BAL
                  TS.IN.IPL
00000000 00000000 0000  1 13:52:35.226  DUMMY1
00000000 00000000 0000  2 13:52:35.226  DUMMY2
00000005 00000003 0000  3 13:52:35.226  DUMMY3
```

You can call `TRACEINTEGER`*x* from within any source file, including user exits. You can place the first `TRACEINTEGER`*x* call at any point in time, provided that it occurs before the point where you want to inspect the trace. These calls do not need to occur during initialization, although that's a good place for them.

A single buffer for storing all of the Integer-Trace information is allocated from the Heap when the first call to `TRACEINTEGER`*x* occurs. If no integers are traced, the buffer is not allocated. The size of the buffer is 12 bytes times the number of trace entries that you specified when you ran `SETTRACE`. If you did not run `SETTRACE`, the default of 250 entries applies.

Because of *when* the Integer Trace buffer is allocated and variables *begin* to be traced, the `APPDUMP` interpretation for entries prior to that point in time are undefined and can be misleading. Undefined values usually show zero. That does not mean that they were zero at that point! Zero is the interpretation provided when the values were not traced yet. All values for a traced variable that occurred *after* the point in time when tracing for that variable began are correct.

Because you can only trace three integers, all `TRACEINTEGER`*x* calls after the third are ignored. Choose your variables wisely.

Do not attempt to trace an element of any type of array, or a string variable. This trace will not show you what you want, and could even cause an abend.

You must call the correct SUB based upon the size of the Integer variable you are tracing. For example, to trace an `INTEGER*2` variable, you must call `TRACEINTEGER2`, rather than either of the other two subroutines. If you mix types, the compiler will implement your call by copying the current contents of the traced variable to a temporary location, and then passing that temporary location to the trace subroutine. You will not get what you want. `APPDUMP` will usually report `(Not Found)` as the variable name if you make this type of calling error.

In similar fashion, do not pass a stack variable to a `TRACEINTEGER`*x* subroutine. Those variables are stored in a section of the stack that is constantly reused for any purpose. `APPDUMP` will usually report `(Not Found)` for the name of those variables if you make this type of calling error.

Variables that are not `GLOBAL`, but are declared outside all functions or subroutines can be reliably traced but will not be reported by name unless you furnish a CDV file. `APPDUMP` will report `(Not Found)` as their name if you furnish a SYM file. Use of a `GLOBAL` name is preferable, because there can be more than one instance of any given name for a local variable. If you can make them `GLOBAL` (even for just this test), do so.

Traced variables are captured only at the points in time when each BASIC function or subroutine is called. `APPDUMP` may show that the value of a traced variable is the same at the beginning of two adjacent function or subroutine calls in the trace, but that does not prove that the value never changed between those calls. Remember, this is a trace and not a trap. If a variable changed many times during the run of a single function and was set back to its prior value before the function returns, the trace could lead you to believe that the variable never changed in value, which is not true for this example.

`GOSUB`s and certain non-BASIC subroutines such as `SUBSTR` and `MID$` do not add an entry to the Application Call Trace, so therefore integers are not traced at those points either. If you need better resolution, add dummy calls before and/or after places where you think a problem might be occurring.

Furthermore, if a traced integer is set to a bad value within a function or subroutine, and a dump occurs before the next call, this trace will not show the bad value, and will not help you to know what line of code caused the problem. It will, however suggest that the current function or subroutine caused the problem, not some other function prior to it. As suggested above, you can modify your source code by inserting dummy calls at test points to help narrow down the failure point when necessary.

For simplicity of implementation, four bytes are saved for all traced integers, even those of size 1 or 2. This **will cause an abend** immediately if an integer being traced is too close to the end of the Data selector (extremely rare but possible). If an abend occurs, add an `INTEGER*4` variable after the point in the source code where that variable is declared to circumvent the abend.

`TRACEINTEGER`*x* routines must be declared `SUB`s and called as shown in the examples above. `FUNCTION` calls do not pass addresses of the arguments, but rather addresses of temporary areas.

Integer Tracing is added only to the Terminal Runtimes SB286LT.L86 at this time. Version 61 or later (February 2009) is required.

2.6 SWI Disable Count – Terminal and Controller

The value of the SWI Disable counter can be interrogated by the application. See heading "SB286LT.L86 Version 54 diagnostic improvements" in the KnowledgeBase article "Why do programs abend with ERRN = 80004186". Here is an example of use.

```
FUNCTION SWIDISABLECOUNT EXTERNAL
  INTEGER*2 SWIDISABLECOUNT ! #times disabled
END FUNCTION
   …
! we expect SWIs to be enabled here.
WHILE SWIDISABLECOUNT > 0
  ! cause a hang so that this error condition can be trapped
  ! with a manual dump.
WEND
```

The SWI disable count also appears in each entry of the Application Call Trace, and can be inspected from a dump.

2.7 Timer Resolution on 4690 V6 "Enhanced" – Terminal

The default Timer resolution reported in the Call Trace is one millisecond for 4690 V6 "Classic" and for all versions prior to V6. In "Enhanced" mode, timer resolution is about ten milliseconds per tick. This large granularity can make timing issues difficult to solve.

To activate millisecond reporting, set the number of trace entries to 4609.

```
SETTRACE T EAMTS10L.286 4609
```

WARNING - do **not** use this option in a 4690 "Classic" environment.

2.8 Heap Error Scanning – Terminal

Application programs can damage the Heap structures maintained by the CBASIC Runtimes. When that occurs, the contents of strings or array elements of any type can be damaged, or the CBASIC Heap Manager can abend while accessing or modifying a string or an array element. You can use this feature to help you determine where heap-damaging code exists.

When activated, this check is called at the beginning and at the end of each CBASIC Subroutine or Function. If an error is found, the check causes an abend. You can review the call trace from a dump to determine what function or subroutine was in control when the damage occurred. To activate this checking, set the number of trace entries to 4610.

```
SETTRACE T EAMTS10L.286 4610
```

An alternate way in which you can do this type of check is to modify your source code around the point where you think an error is occurring. Add this declaration and calling code:

```
SUB SCANFREEHEAPBLOCKS EXTERNAL ! Heap damage?
END SUB
  ...
CALL SCANFREEHEAPBLOCKS ! dump if heap damage
```

2.9 Lock and Unlock Tracing – Terminal

It can be difficult to determine where or why an IO Processor Lock or Unlock occurred. This new trace captures important Lock and Unlock information in the Call Trace.

To activate Lock and Unlock tracing, set the number of trace entries to 4616.

```
      SETTRACE T EAMTS10L.286 4616
```

Here is sample output from `APPDUMP`:

```
2689 12:36:57.688 0167:0052 4A 00   3    JAVAEVENT
2690 12:36:57.688           4A 00   0 Runtimes trace: 0020 7004
2691 12:36:57.688 00f7:2331 4A 00   1  TSUPEC51
2692 12:36:57.688 0477:cbb3 4A 00   2   TSU51.FB
2693 12:36:57.688           4A 00   0 Runtimes trace: 0020 000A
```

The line at entry 2690 shows that for session #32 (0020H), a `LOCKDEV` (7xxxH) occurred along with a `PURGE` (xxx4H). The line at entry 2693 shows that for session #32 (0020H), an `UNLOCKDEV` (0xxxH) occurred and transitioned to state #10 (x00AH).

2.10 Heap Allocate and Free Tracing – Terminal

This new trace captures the address of each Heap block allocated or freed by the CBASIC Runtimes. The trace is reported from a dump, near the end of the `APPDUMP` run.

The last sixteen addresses are saved and reported. To see this trace in a dump, you must specify the `/dh` option when running `APPDUMP`.

To activate tracing for Heap Allocates and Frees, set the number of trace entries to 4624.

```
      SETTRACE T EAMTS10L.286 4624
```

Here is a sample of reporting by `APPDUMP`. To analyze this trace, work from the wrap point backward. For each address, find that memory block in the Heap interpretation and examine its contents to determine if it contains what you expect it to contain.

The Nest= field is the call nesting level associated with the depth of function or subroutine calls (00 signifies "in the mainline").

```
HeapTrc addr of entries=051F438C
Num entries=16
  Entry    0:  Nest=01       Free          05DFCBDC
  Entry    1:  Nest=00   Allocate 05D7DFF8
  Entry    2:  Nest=00       Free          05D7DFF8
  Entry    3:  Nest=03   Allocate 05E7966C
  Entry    4:  Nest=02       Free          05E7966C
  Entry    5:  Nest=03   Allocate 05E7966C
  Entry    6:  Nest=04   Allocate 05E79F9C
  Entry    7:  Nest=03       Free          05E7966C
  Entry    8:  Nest=FF   ************** wrap **************
  Entry    9:  Nest=01   Allocate 05E722C8
  Entry   10:  Nest=01       Free          05E791A4
  Entry   11:  Nest=02   Allocate 05DFBEA4
  Entry   12:  Nest=02       Free          05E722C8
  Entry   13:  Nest=02   Allocate 05DFCBDC
  Entry   14:  Nest=02       Free          05DFBEA4
  Entry   15:  Nest=01   Allocate 05E791A4
```

2.11 Tracing of Runtime Entry Points – Terminal

This capability was added with change CBRT066 to the Terminal Big Memory Runtimes SB286LT.L86 in February 2011.

For selected Entry points in the Runtimes, this new trace captures the address of each Entry point, along with the address of its caller. This trace generally is most helpful when you are getting an ABEND in a store from your application, and you are able to capture a dump at the point of the error or shortly thereafter. It helps you to find the specific statement which is causing the abend. The trace is interpreted by APPDUMP, prior to the reporting of the Open sessions.

This trace is normally OFF, because it can typically consume 20% more processor time when it is activated. If your terminals are running slowly because of the processor load on each, you might **not** want to use this trace unless you are willing to accept the additional processor load that it demands. If your terminals are lightly loaded, activation of this trace might be unnoticed by the operators.

To activate this trace, relink your Terminal Sales application with version CBRT066 or later of the Runtimes, and set the number of trace entries to 4640. Then reload that new application in the Terminal.

```
SETTRACE T EAMTS10L.286 4640
```

The last 256 calls to the Runtimes are captured in a wrapping buffer that is allocated from Heap memory. Using the output from APPDUMP and an Interlisting of the module in question, you can compare the source code and the trace to determine what Runtime functions were called, and whether or not an error occurred during the execution of each.

Here is the last part of a sample trace, as reported by `APPDUMP`. This was a ctl-alt-del dump that was captured at an arbitrary point during initialization.

```
Runtime Entry Point trace contains 256 entries:
count  time         ERR  Nest  Caller -> Called
 …
 241                 1A   1  TRR.INIT + a (0287:0193)  calls  _cb_prolog (0067:0000)
 242                 1A   2    TRR.INIT + 362 (0287:04EB)  calls  _cb_epilog (0067:02CF)
 243  11:13:53.218   1A   1  EAMTRACE + a (02F7:0226)  calls  _cb_prolog (0067:0000)
 244                 1A   2    EAMTRACE + 20 (02F7:023C)  calls  o.rsubs1 (0077:004A)
 245                 1A   2    EAMTRACE + 46 (02F7:0262)  calls  o.rsubs1 (0077:004A)
 246                 1A   2    EAMTRACE + 68 (02F7:0284)  calls  _cb_epilog (0067:02CF)
 247                 1A   1  EAMTRACE + a (02F7:0226)  calls  _cb_prolog (0067:0000)
 248                 1A   2    EAMTRACE + 20 (02F7:023C)  calls  o.rsubs1 (0077:004A)
 249                 1A   2    EAMTRACE + 46 (02F7:0262)  calls  o.rsubs1 (0077:004A)
 250                 1A   2    EAMTRACE + 68 (02F7:0284)  calls  _cb_epilog (0067:02CF)
 251  11:13:53.218   1A   1  INITBONUSOPTS + a (027F:05FE)  calls  _cb_prolog (0067:0000)
 252                 1A   2    INITBONUSOPTS + 21 (027F:0615)  calls  o.dim.basic (011F:000B)
 253                 1A   2    INITBONUSOPTS + 42 (027F:0636)  calls  o.dim.basic (011F:000B)
 254                 1A   2    INITBONUSOPTS + 5f (027F:0653)  calls  c.set.d1 (006F:009D)
 255                 1A   2    INITBONUSOPTS + 7b (027F:066F)  calls  c.set.d1 (006F:009D)
 256                 1A   2    INITBONUSOPTS + 96 (027F:068A)  calls  _cb_open + 8 (00FF:075E)
```

Each trace entry is numbered from 1 through 256, with entry 1 the oldest and 256 the newest. The time stamp is shown for every eighth trace entry. The ERR field is a cumulative (hex) count of the number of ON ERROR occurrences at the point when that trace entry was captured. It wraps to 00 after FF. The Nest level is the number of function calls deep on the stack at that point. APPDUMP then shows the symbolic name and its selector:offset address in *your application* where that entry point was called, and finally the symbolic name and its selector:offset address in the Runtimes that your application is calling.

Notice that the time stamp does not appear on every trace entry. Getting the time stamp is much more expensive than capturing the called address. To minimize performance degradation, only one in every eight calls is time stamped. This may not be a problem for most cases because there often are many more than eight calls completed within a single timer tick.

Only the actual subset of all Runtime Entry points that are *definitely* called by *TGCS SA* Base code and its *TGCS SA* Features are traced. In the interest of minimizing Code space requirements and Runtimes change, additional Entry points in the Runtimes that are implemented but which SA does *not* reference, are not traced. NRSC enhancements or your unique add-ons could use those functions. If that is the case for a dump you are analyzing, you will have to factor that into your analysis. If specific Runtimes functions are missing and that makes your analysis difficult, you can request that TGCS consider adding those individual functions to this trace. *You will have to specify the exact names of every Runtimes function you want added. TGCS makes no commitment to add those functions.*

While interpreting the trace shown above, find the last entry or the entry of concern. Entry number 256 (in the excerpt above) is a call from `INITBONUSOPTS + 96 (027F:068A)` to Runtimes function `_cb_open + 8 (00FF:075E)`. `INITBONUSOPTS` is in file EAMTS2BC. Here is a subset of the Interlisting for EAMTS2BC that shows offsets corresponding to entries 251 through 256 of the trace excerpt that is shown above.

```
2085:SUB InitBonusOpts PUBLIC
label_26:
        05f4 : 1e                PUSH            word ds
        05f5 : c8100000          ENTER           word 010h, byte 00h
        05f9 : 8966fe            MOV             word [fffe+bp], sp
        05fc : 6a00              PUSH            byte 00h
        05fe : 9a00000000        CALL            _cb_prolog                 (entry 251)
2086:    STRING    work1$
2087:    INTEGER*1 errFlag
2088:
2089:! AIRENH02 Option initialization for bonus buy
2090:!  For redoing a transaction after reloading
2091:   DIM BB.TOTAL(5)
        0603 : 6a05              PUSH            byte 05h
        0605 : b001              MOV             byte al, 01h
        0607 : 98                CBW
        0608 : 50                PUSH            word ax
        0609 : 6a04              PUSH            byte 04h
        060b : b004              MOV             byte al, 04h
        060d : 98                CBW
        060e : 50                PUSH            word ax
        060f : c41e0000          LES             word bx, [0*]
        0613 : 06                PUSH            word es
        0614 : 53                PUSH            word bx
        0615 : 9a00000000        CALL            o.dim.basic                (entry 252)
        061a : 06                PUSH            word es
        061b : 53                PUSH            word bx
        061c : 8f060000          POP             word [0*]
        0620 : 8f060200          POP             word [2*]
2092:   DIM BB.COMPETITORS(5)
        0624 : 6a05              PUSH            byte 05h
        0626 : b001              MOV             byte al, 01h
        0628 : 98                CBW
        0629 : 50                PUSH            word ax
        062a : 6a04              PUSH            byte 04h
        062c : b058              MOV             byte al, 058h
        062e : 98                CBW
        062f : 50                PUSH            word ax
        0630 : c41e0000          LES             word bx, [0*]
        0634 : 06                PUSH            word es
        0635 : 53                PUSH            word bx
        0636 : 9a00000000        CALL            o.dim.basic                (entry 253)
        063b : 06                PUSH            word es
        063c : 53                PUSH            word bx
        063d : 8f060000          POP             word [0*]
        0641 : 8f060200          POP             word [2*]
2093:!  Get the Bonus Buy Options and Descriptors
2094:!  Open the BB Options File
2095:   N% = 0                                  ! Init disk error flag
        0645 : c70600000000      MOV             word [0*], 00h
2096:   C$ = " "                                ! Init for SUBSTR
        064b : 0e                PUSH            word cs
        064c : 683b00            PUSH            word 03bh*
        064f : 1e                PUSH            word ds
        0650 : 680000            PUSH            word 00h*
        0653 : 9a00000000       CALL            c.set.d1                (entry 254)
2097:   BB.SUMMARY.ON = 0
        0658 : c606000000       MOV             byte [0*], 00h
2098:   BB.TYPE.ON = 0
        065d : c606000000       MOV             byte [0*], 00h
2099:   BB.ENABLED = 1                         ! DEFAULT TO ENABLED
        0662 : c606000001       MOV             byte [0*], 01h
2100:   TS.TS11WERR$ = ""
        0667 : 0e               PUSH            word cs
        0668 : 682600           PUSH            word 026h*
        066b : 1e               PUSH            word ds
        066c : 680000           PUSH            word 00h*
        066f : 9a00000000       CALL            c.set.d1                (entry 255)
2101:   TS.ER.RETURN = 0                     ! Log errors
        0674 : c70600000000     MOV             word [0*], 00h
2102:   OPEN "R::EAMCOMPP" DIRECT RECL 843 AS 64  NOWRITE NODEL
        067a : 681803           PUSH            word 0318h
        067d : 6a00             PUSH            byte 00h
        067f : 6a01             PUSH            byte 01h
        0681 : 6a40             PUSH            byte 040h
        0683 : 684b03           PUSH            word 034bh
        0686 : 0e               PUSH            word cs
        0687 : 683e00           PUSH            word 03eh*
        068a : 9a00000000       CALL            _cb_open                (entry 256)
2103:   IF TS.TS11WERR$ = "" then begin
```

3.0 Dump Analysis – APPDUMP

APPDUMP is used frequently by TGCS application support personnel to do problem analysis. It is being made available to the retailer and Business Partner community. This tool will assist programmers in isolating the cause of problems occurring in their version of the CBASIC applications.

APPDUMP provides considerable information to assist programmers. The main sections are the state (value) of the global variables when the dump occurred, the application call trace, and the application stack. Much more is provided. The stack shows the most recent calls to functions along with their parameters.

This dump tool provides the following for both Controller and Terminal dumps:

1. Listing of the application's global variables along with their values at the time of the dump
2. The stack, for programmers' use
3. Files opened by the application and some of the data read or written to the file
4. Event bits (tells if you have a missing resume in your `ON ERROR` routine)
5. A list of processes running at the time of the dump
6. A list of recent runtime errors encountered
7. If using the trace runtimes, the listing of the CBASIC called routines for the specific application, subject to the limits you chose (5200 max, 250 default)
8. Consoles (some shots of the text screens on the controller)
9. (only in the Controller) A disk trace for the controller

APPDUMP requires a symbols file (*.SYM) or a codeview file (*.CDV) that *matches* the code level of the application running in the dump. Use of a CDV file, if available, is preferred. To help you with managing the correct SYM file, consider using the `-m` option of `POSTLINK`, as described on page 24.

APPDUMP is somewhat fragile. It can abend if you provide a SYM or CDV file that does not match the dump, or if the dump is incomplete, or if something in the dump occurs that it does not recognize correctly. The tool is furnished to you **as-is**. Support is on a best-effort basis -- TGCS is not obligated to fix any problems you report, and you should set your expectations accordingly. However, if you have a support contract for SA or GSA and find a "complete" dump that this tool does not process, and you are sure that the SYM or CDV file matches, please e-mail the output of the CPAD create problem analysis diskettes utility along with the matching SYM or CDV (symbols file) to rssdoc@us.ibm.com and send an e-mail to cmessi@us.ibm.com alerting us that you have sent a dump, what the files were named, and what the error message was that you received (if any).

To obtain correct dump analysis, it is absolutely essential that the matching version of a SYM or CDV file is available. If unavailable, correct analysis might be impossible. You should assume that the SYM and CDV files will change with *every* relink of your application. Every time you send a new version of your application to a store, you should always send the matching SYM or CDV file to that same store. When furnishing a dump to TGCS for analysis, or when retrieving it for your own review, always retrieve and furnish the SYM or CDV file from the same store as the dump. **Do not obtain it from a different store or from your headquarters system.** Submitting the wrong SYM or CDV file will delay problem resolution. As stated above, consider using the `-m` option of `POSTLINK`, as described on page 24.

Some customers choose to package the SYM file into the 286 file by using the `-c` option when post linking the application. Please see knowledge base article "Postlink modification to include the application symbol file in the executable" for further information. A potentially better option is to use the –m option of `POSTLINK`, as described on page 24.

How do you know whether you have a complete and 'valid' dump? Verify that the dump got to the controller or the controller came back up after the dump. Verify that the OS tool Auto Dump Analyzer processed it successfully. The output of this automatic process is kept in ADX_SDT1 on the controller that the dump was taken on or the one that the terminal is attached to. The file names are ADXEnnCF.DAT or ADXEnnTF.DAT (where *nn* = the node name, such as cc, and *CF* = controller and *TF* = terminal dump.) Read that file and determine if the date and time match the time you believe the dump occurred. If it appears to have valid output, then the dump is probably valid.

The input to `APPDUMP` is fairly straightforward. Here is sample output of `APPDUMP` during its initialization:

```
  ##############################################
  # APPDUMP: 4690 BASIC Controller or Terminal #
  #          Big Memory Model Dump Formatter   #
  #                   02-23-2009               #
  ##############################################
```

```
Name of dump file (default adxcsltf.dat):  m:\h60csltf.dat
```
Note: This is where you point to the appropriate dump name and location.

```
Name of BSX file (default adxrt8gl.bsx):  m:\h60rt8gl.bsx
```
Note: You need to point to the kernal BSX file along with its location. For terminal dumps, use adxrt8gl.bsx (adxrt1sl.bsx for 4683 terminals). For controller dumps, use adxct8sl.bsx.

```
Name of SYM or CDV file (default EAMTS10L.SYM.
Enter 0 if none):  m:\h60ts10l.sym
```
Note: Depending on which application was running at the time that you need to see information about, point to the appropriate sym file.

```
Application process name (default mpostapl.
Enter 0 if none):  mpostapl
```
Note: Again, enter the name of the running application that you want to review. (use rpostapl for a mod 2 terminal application.)

```
Max number of array elements shown (1 - 120,
default = 25):  120
```
Note: In a given Global Array, how many elements should be listed.

```
Max length of string shown (default = 512):  512

Name of output file (default "SCREEN"):  dump.out
```

Output shown on next page.

```
   Processing the SVC trace...
    Application details..........
    Checking the application heap...
    Processing the application call trace...
    Processing application variables............
    Processing the stack
    Processing open files...
    Processing pipes...
    Processing console...

  Shall I format the serial device driver data?  n
    OK!
```

For ease of use, you may prefer to run `APPDUMP` with a BAT file. For example, the following set of BAT file commands will run `APPDUMP` on the last controller-terminal dump and interpret the results for the SA Terminal Sales program that was running in the Terminal section of that controller. The results are saved in a file so that they can be retrieved to another system, and then they are displayed for you to review.

```
define bsx=c:\adx_spgm\adxct8sl.bsx
define sym=c:\debug\eamts10l.sym
define apl=mpostapl

c:\appdump c:\adxcslcf.dat %bsx% %sym% %apl% 0200 6500 c:\tdump.out /d
xe  c:\tdump.out
```

In the rare case where there are two or more processes with the same name, you can specify which one to interpret by prefixing the process name with a numeric character. Use '1' for the first of that name, '2' for the second, etc. For example, if there are two processes named `EAMDMMLL`, specify `2EAMDMMLL` if you want the second one interpreted. If you specify `EAMDMMLL` (or `1EAMDMMLL`), the first one will be interpreted.

3.1 APPDUMP Crashes. Is that Tool broken?

Crashing of `APPDUMP` is *usually* caused by a mismatching SYM file, although a program bug is not out of the question. It is especially difficult to know which of the two is at fault if `APPDUMP` crashes before it can perform the testing necessary to know that the message `The SYM file is NOT a good match` should be given. In any case, first verify that you have the correct, matching SYM file. If you are certain that it matches, furnish your dump and SYM file to TGCS for analysis. Please read the license agreement beginning on page 31. *TGCS does **not** warrant that your problem with `APPDUMP` will be fixed.*

To help you with managing the correct SYM file, consider using the `-m` option of `POSTLINK`, as described on page 24.

3.2 What Application is Running?

With a Terminal dump, the application that is running at the time of the failure is usually the TGCS SA Terminal Sales Application (EAMTS*xx*L.286), or TGCS GSA Terminal Sales Application (EALTS*xx*L.286).

With a Controller dump, the failing application is often Checkout Support (e.g., EAMCSMLL) or a report (e.g. EAMRPCRL).

To determine what application is running, look at the `APPDUMP` output for lines like this:

```
  Status PID  Function Process   Priority     Application
   (W)   0008  appl     mpostapl1   196    R::ADX_IPGM:EAMTS11L.286
```

The key things are

- (W)=waiting, (C)=Current, (R)=Ready to Run.
- mpostapl*x* is the main terminal application

If the current process is ADXTST1, it means that the operator issued a request at the controller to dump the terminal. ADXTST1 is merely fulfilling that dump request. For this case, if exactly one application is marked (R), you should consider it to be (C). If more than one of them are marked (R), any one of them could have been (C) at the time of the dump request – guess; perhaps the ASR/SVC trace will help for this case.

If there are one or more processes that are marked (R), it is likely that the process marked (C) is in a hard loop and is not relinquishing control.

3.3 What is the general health of the Application?

For push-button dumps and for dumps that are forced by the operator from the controller, you will usually see the status for TGCS-SA and TGCS-GSA as (W). This is usually the case because both SA and GSA have a main loop that simultaneously waits on a timer (usually 1 second), the keyboard, the scanner, the MSR, and (as needed) a serial port. When an event occurs or the timer pops, the required actions are usually performed relatively quickly, and the main loop immediately issues another wait. Therefore, it is far more likely that the application will be marked (W) instead of (C).

When the terminal sales application is marked (C), it is because of one of these conditions:

1. The application has much to do since it was last dispatched, or
2. There was a trap on a specific runtime error (forcing a dump), or
3. A statement caused an immediate abend (rare), or
4. luck

A terminal sales application can also be marked (R), which means 'ready to run'. This happens when it is not waiting for any events, but some other process is currently active and that process may have caused the dump. Report this problem to the team who maintains that process.

From the viewpoint of the Cashier, a terminal can appear to be "hung" for a variety of reasons. A hang can show a status of (W), (C) or (R) for the application, although (W) is usually indicated. Here are *some* examples of conditions that will appear as a hang, along with the most likely state for each condition.

1. (W) the application is currently prohibiting a <CLEAR> because it is waiting for the pinpad, a host credit pipe response, or some other task that must not be interrupted,
2. (W) the IO Processor is deliberately disabled in its current state while waiting for an event,
3. (W) a User Exit is in control, waiting for an event,
4. (C) it is in a hard loop (rare),
5. (W) a `READ`, `WRITE` or `GETLONG` that should normally be immediate does not complete,
6. (C) an `ON ERROR` routine is repeatedly called because it does not properly handle a specific error,
7. (W) a Manager Override is required,
8. (W) there is a problem with the Input Sequence Table,
9. (C) a required file is inaccessible, and there is no `ON ERROR` code to handle that situation correctly,
10. (R) another application or an OS driver is preventing SA from running.

<u>The SWI Disable Count</u>

Look for a line like this:

```
SWIs disabled 1 time(s)   {or}
SWIs are enabled
```

During normal straight-line code, SWIs are almost always enabled. There are very few exceptions, and apply only for cases whereby the application is designed to perform a non-interruptible set of actions. One such case is when updating hard totals and the totals save file. Another, very brief case is when a CBASIC string is being assigned.

SWIs are always disabled while processing an `ON ASYNC ERROR`, while processing an `ON ERROR`, for certain parts of string manipulation, and while processing a `TCLOSE`. (There may be other cases as well.)

If an `ON ASYNC ERROR` routine or an `ON ERROR` routine returns improperly, the SWI disable count will be left at a positive value. This will eventually cause an abend, because printer operations in that state will cause the application to run out of event bits. See KnowledgeBase article "Why do programs abend with ERRN = 80004186?". The SWI disable count is traced in the SUB Application Call Trace.

Open File and Pipe Sessions

Look for a line like this:

```
Open sessions   (trailing zeros are not formatted):
```

The items that follow are the I/O buffers for each session that is open. For fixed-length records, this buffer shows the data that was last read or written. For variable-length records (such as for the printer), interpretation can be tricky because you might be seeing data from several different reads or writes. A few bytes from the most recent read or write can appear at the beginning of the buffer; followed by *some* of the data from a previous read or write.

For SA, the following sessions might contain information that is useful for debugging the current problem. Note that these buffers are not time-stamped, and data could be immaterial to the current problem.

- 4 - $AMITEMR. This contains the data for the item that was last read from the item record file using session #4.
- 20 - EAMSDESC. This is the last 2x20 descriptor read from the session #20 file. Because of buffering, it is not necessarily the last 2x20 message displayed.
- 32 - IOPROC. This describes the previous and current states, the last key pressed, and whether or not an I/O Processor error occurred. See the BASIC Reference manual, "Input/Output Processor" section, for a description of how to interpret this data.
- 34 - CR. This contains data for the last line that was printed on the printer receipt paper.

The Call Trace

Look for a line like this:

```
SUB Application Call Trace (listed from oldest to newest):
```

This is the most useful section to review. It shows the names of the functions that were called immediately prior to the dump. Time stamps help to correlate specific errors with the CBASIC Error trace and with other parts of the dump. The 'Level' column indicates the nesting level of function calls at that point. Indentation also shows nesting level. With Runtimes version 55 and later, the `ON ERROR` count and the SWI Disable counter are traced, instead of BP.

At the suspected point of the error, if the names of the functions are not part of TGCS-supported code, then the customer, not TGCS, is responsible for diagnosing the cause of the error.

The default size of this trace is usually 250 or 400 entries. You can use SETTRACE to increase the number of entries when the default size is insufficient. The maximum size you can specify is 5200.

If the correct, matching CDV file is available, use it instead of the SYM file when you run `APPDUMP`. Local function/sub names and all local variables are resolved, which is a great help in debugging problems. You need a 4690 Application Debugger license in order to be able to build CDV files.

Compare the time of the last entry against the time of the dump. If they are far apart, the application is in a hard wait or hang, or is being prevented from running by the OS or a higher-priority task,

The Application Stack

Look for a line like this:

```
      SUB Application Stack:
```

Find the entry at offset (0000). The contents there should be the same as the application's DS register. Then look after that line for the first line that does *not* contain 5353. The address of that line is the number of bytes of stack that were *never* used. If this address is relatively small, this application is dangerously close to experiencing a stack fault, and one could occur in the future. There probably isn't a stack problem with *this* dump. The maximum stack is specified with linker option `[STACK[MAX[FFE]]`. The maximum stack is usually best for all applications, unless memory is limited. **Do not use a value larger than FFE when linking your application.** Larger values will allow the stack to wrap, which will cause an ABEND at some point *after* the wrap occurs. It is better to catch the failure at the point where it tries to wrap.

<u>What CBASIC Runtime Errors were Traced in Memory?</u>

Look for a section like this:

```
    The last 15 errors recorded by the runtimes:

      FU 00000022   05-09  12:01:34.419   session   41
                  ON ERROR at 0207:30f7  main + 1abf

      OE 80A50009   05-09  12:01:34.419   session   41
                  ON ERROR at 0207:30f7  main + 1abf

      SS 0000006E   05-09  12:01:34.421
                  ON ERROR at 0207:30f7  main + 1abf

      RN 0000007A   05-09  12:01:34.823
                  ON ERROR at 0207:30f7  main + 1abf
```

This is also referred to as the CBASIC ERROR TRACE. There will usually be 15 errors traced in this section of the dump. If fewer than 15 appear, the application is well-behaved or is just starting to initialize. They are reported in chronological order. See the CBASIC Reference manual ("Appendix B. Runtime Error Messages"), and/or the 4690 Messages Guide for a description of what "OE 80A50009" and "SS 0000006E" mean.

Compare the time of the last entry against the time of the dump. If they are at about the same point in time, that last error (or the error immediately preceding it) may have setup the conditions to cause the dump.

Duplicate, consecutive errors are combined so as to make the best use of the limited trace of 15 errors. If an error is repeated, the time stamp shown is when the *first* error of that type occurred. The times of the second and subsequent errors of that same type are not precisely traced; they are at that same instant of time or later.

```
      OE 00000052   07-28  09:14:53.112                  repeated 6 times
                  ON ERROR at 0307:6343  main + 123f

      OE 00000052   07-28  09:14:56.594                  repeated once
                  ON ERROR at 02ff:872d  TS1SEC02 + 50e9

      CU 0000000E   07-28  09:14:57.577   session   61
                  ON ERROR at 0307:6343  main + 123f
```

The `ON ERROR` routine of the application typically catches all errors and *is supposed to* react appropriately for each expected error. For example, a FU 00000022 error on session 41 is typically corrected in the `ON ERROR` routine by opening the file that session 41 is supposed to use and then doing a `RESUME RETRY`.

Some errors are fatal, and usually occur due to an error in program logic or missing logic to handle unexpected, rare errors. It is possible, but very unlikely that the `ON ERROR` routine is designed to recover from these errors. *Some* of these fatal errors are:

AE 00000004	- an array being accessed is not dimensioned.
BW 80840017	- an IOPROC Read occurred while the driver is locked.
RN 0000007A	- a `RESUME` was attempted, but the action is not resumable.
SS 0000006E	- a parameter of `MID$` is zero or negative.
SU 0000007E	- an array subscript is out of bounds.

The `SETTRAP` tool can be used to force a dump on any type of error, such as SU. This preserves the stack and calling trace so that the source of the error is more easily identified. See page 25 for more information on `SETTRAP`.

If you do not have the luxury of using `SETTRAP SU` and attempting to recreate the issue, a more difficult method can be used to further analyze this type of error using the current dump:

1. Note the time of the SU error
2. Look through the call trace to find out what function or subroutine was running at that point in time. If the trace does not contain that point in time (because it wrapped), you cannot proceed. If many functions have that same time stamp, it will be very difficult to proceed.
3. Review the source code from the above step to determine what code in it may have caused that type of error. If source code is not available (customer code?), or if there are too many places to consider, you might not be able to proceed.
4. If `GLOBAL` variables were referenced at the potential error points, examine their contents to help confirm where the error could have occurred. If their values changed since the error occurred, it will be very difficult to proceed. If local variables were used, you cannot proceed unless you can determine from `GLOBAL` variables how those local variables were set.

Sometimes, an application is in a restart loop. If you see more than one RN error, the application is in a restart loop. It attempted to restart, but because of residual content of variables or recovery data, a fatal error occurred again. Refer to the first RN error to try to determine what started the looping. Often, you will not see this first error because the trace wrapped, so you can't easily determine if a restart occurred. A `SETTRAP` dump may be necessary to find the first RN error. Alternatively, the Application Event Log (if available) will have all unexpected errors if the `ON ERROR` routine logs them.

IO10105 added `RESTART.TRACE$` and `RESTART.LAST.ERR$` to SA in January 2009 to help you determine what errors are causing a restart. Examine the contents of those `GLOBAL` variables.

Finding the Point of the Error

CS:IP gives you the location of the current instruction at the time of the dump. Although this is precise, it usually is grossly inadequate. Ideally, you would like to know the name of the current function and the specific CBASIC source statement that is associated with that CS:IP. An interlisting is usually required to precisely identify a source statement, but you can often make a reasonably good guess even without it.

If a trap occurred, the Sub Application Call Trace shows the names of the functions called immediately prior to that error. An example is shown below. In this example, `TSS3EC01` called `?SET.EMOD$`. `TSS3EC01` then called `DATETIME`. `DATETIME` called `?GETTIME$`, `?GETDATE$` and `CR.LINE.SPACING`. `TSS3EC01` then called `USER.EXIT`. `USER.EXIT` called `?SET.EMOD$` and then called `PRINT.EXIT$`. The dump occurred while `PRINT.EXIT$` was executing. The current function is determined by finding the last function that has a nest level equal to the current value of nestlevel. In this example, nestlevel is 3, so `PRINT.EXIT$` is executing. If nestlevel was 2, then `USER.EXIT` would be executing. If nestlevel was 1, then `TSS3EC01` would be executing. The nest level (and indentation) is incremented at each call and decremented at each return.

```
17   01f7:435e  ff26   1    TSS3EC01
18   0317:0128  feaa   2     ?SET.EMOD$ + 5
19   01f7:34de  feac   2     DATETIME
20   01f7:064e  fe56   3      ?GETTIME$
21   01f7:071d  fe56   3      ?GETDATE$
22   01f7:2e04  fe56   3      CR.LINE.SPACING
23   01f7:1739  feac   2     USER.EXIT
24   0317:0128  fe52   3      ?SET.EMOD$ + 5
25   049f:0013  fe4c   3      PRINT.EXIT$ + 5
   nestlevel is currently at 3 (in the SUB/FUNCTION shown above):
```

If a function name is shown with a (hex) offset, e.g. `DATETIME` **`+ 3F4`**, it signifies that that function is local, i.e. private (not public). When using a SYM file, `APPDUMP` cannot resolve these local names because they are not included in the SYM file. If you want to know their names precisely, you must use either a CDV file instead of a SYM file, or you must refer to an interlisting to find the name at that symbol+offset. You could also make frequently-called functions `PUBLIC` so that their names appear in the call trace of future dumps.

You could also review the stack to determine the name of the current function. Here are the important lines of the stack for the example above. Note that a stack grows downward, so the calls are listed in reverse order.

```
(fe50)  1f7:19cd  USER.EXIT + 28f calls PRINT.EXIT$ (49f:000e)
(feb0)  1f7:53da  TSS3EC01 + 1079 calls USER.EXIT (1f7:1739)
(ff2a)  317:51ee  main + 54 calls TSS3EC01 (1f7:435e)
```

If the current function is relatively small, you can usually guess which source statement is failing by finding the last traced error and by looking in the current function for a statement in the source code that could cause that error. For example, if the last error was SU 0000007A (array subscript), look for a statement that accesses an array element. If there is more than one of these, look at the current IP to determine how far into that routine the error occurred. Refer to the contents of variables to help narrow down your search.

If you need to precisely resolve the point of error to a specific CBASIC source statement, and the above methods do not work, you will probably need an interlisting.

3.4 Obtaining an Interlisting

Here is a BAT file used by TGCS L3 support to create an interlisting for an SA file. Use it only as an example and adapt it to your development environment. Note: there are only two statements here, SET and BASIC. The SET BASINC… is all on the same line!

```
SET
BASINC=s:\satof11DB\latest\source;s:\saefte11DB\latest\source;s:\sa461012D
B\latest\source;s:\saem11DB\latest\source;s:\sa11DB\latest\source;
BASIC %1  [IDFBU,M(b),R(xxxxxxxx.lbj)] >interlst.lsu:
```

When reviewing an Interlisting based upon an address from a dump, you might notice that the sel:off from the dump differs by a few bytes as compared to the offset of the equivalent point from the Interlisting. If your Interlisting truly matches the level of code running in the dump, the discrepancy is probably due to the attempts by the Linker to combine files on any single-byte boundary rather than a 16-byte paragraph boundary*. That behavior is not a bug, it is an attempt to generate the smallest possible 286 file.* If you do not like that behavior, and you have a few KB or more available in the 2 MB code space, you can do these three things to ensure that the offsets match exactly:

1. Obtain the V3.15 (07/08/2010) version of the compiler, BASIC.EXE/286. Recompile all of your BAS files and LIB86 all of your L86 files that you use. This version of the compiler always increases the size of the code for every BAS file to an exact multiple of 16 bytes.
2. Obtain and use V 65 of SB286LT.L86 (07/09/2010).
3. If your offsets still do not match, check the size of your ADXMEL0L.L86 file. It should be 21,120 bytes in size.

3.5 SA Global Variables useful for Problem Analysis

Here are some Global variables that may help you to determine from a dump what occurred recently, what is currently occurring, or what should be occurring in TGCS Supermarket Application.

<u>ASYNC Error Tracing</u>

IO06213 added a trace in global string `TS11C.ASYNC.ERROR.TRACE$`, which captures useful information at each ASYNC error.

<u>Key Presses and 2x20 Displays</u>

IO06213 added a global string array `IO.PROC.LINES$` that contains the last `x` lines written to the 2x20 or read from the I/O Processor. `x` defaults to 50 but can be changed in a user exit.

<u>Restart Trace</u>

An unrecoverable error causes a `B000 APPLICATION ABEND` guidance, and results in SA attempting to restart itself after the operator clears the `B000`. IO10105 added a global string `RESTART.TRACE$` to capture the date, time and last error that occurred immediately prior to the `B000`.

<u>Serial Port (Pinpad) Trace</u>

IO04381 added a global string `EE.PINPAD.TRACE$`. Information traced includes a time stamp and the first 110 bytes of each read or write.

4.0 Other Tools

4.1 POSTLINK

This tool modifies a recently-linked 286 file so that it loads faster. A change was made in 2010 to allow *merging* of the SYM file into active code.

A “merge” of the SYM file is different from a “combine”. In previous versions, the `-c` option would combine (append) the SYM file to the end of the 286 file so that you could be certain that the matching SYM file is always available.

The new `-m` option actually expands the last Code segment by approximately the number of bytes required to contain the entire SYM file, and then appends the SYM file at that position following that code area in the 286 file. This not only ensures that the matching SYM file is always available, it also ensures that the matching SYM file is *loaded into memory* and appears automatically in any future dump containing that application. This means that you do not have to furnish the matching SYM file with a dump, because it is already included *in* the dump!

This new `-m` option has a limitation. The amount of space remaining in the last code segment must be sufficient to contain the entire SYM file. If the SYM file is too large to fit, then `POSTLINK` turns OFF the –m option and turns ON the `-c` option for that run. *You will still have to provide the matching SYM file when the size of your application is as stated in this paragraph.*

`APPDUMP` was changed to automatically look for the SYM file in the dump, and uses that data when it is available. Prior to using the SYM information, `APPDUMP` extracts the contents of that area of the dump to file APPD_SYM.SYM, and then uses that file as its SYM. The SYM file you specify is not used and is irrelevant for this case.

4.2 SETTRACE

This tool normally establishes the number of entries in the Application Call trace. The default provided by the Runtimes is usually 250 entries. You can increase the number to a maximum of 5200 entries.

Each saved entry uses 12 bytes of heap memory. The default amount of memory that this tracing uses is 3,000 bytes for 250 entries. This can be increased (or decreased) at your discretion, by running `SETTRACE`. The maximum number of entries is limited to 5,200, which will increase memory requirements of your application by almost 64 KB. If you are using terminals that have little memory to spare, you might need to live with 250 (or even 25) for the number of entries.

After linking a new terminal or controller module with the required SB286*.L86, you can change the default number of trace entries by running `SETTRACE`. Specify the environment that this program runs in (`T` or `C` for Terminal or Controller), the name of your application program, and the number of entries to save. For example:

```
SETTRACE T EAMTS10L.286 5000
```

If it runs correctly, you will see:

```
Found it!  Are you sure you want this file modified (Y/N)?
```

You should type `y` in response to this message. To review these trace entries, you can use either the latest version of the 4690 Application Debugger, or run the Application Dump Analyzer (`APPDUMP`) against a dump.

Relinking undoes the previous `SETTRACE` action, so you should rerun `SETTRACE` after each relink until a problem you are experiencing is fixed.

Some customers choose to make `SETTRACE 5000` a permanent part of their relinking procedure, so that future problems are more likely to provide good documentation with the first occurrence of a dump.

A second use of `SETTRACE` is to turn on certain CBASIC Runtimes tracing features that are normally dormant. The specific range of 4608 through 4863 entries (hex 1200 through 12FF) is otherwise reserved, and is used for this purpose. Run `SETTRACE` and specify the number of entries as 4608 plus the number associated with each of the options (below) that you want to invoke. This capability was first released with Runtimes V64.

```
  1 - 4690 V6 Enhanced timer resolution (x1201)
  2 - SCANFREEHEAPBLOCKS in _cb_prolog and _cb_epilog (x1202)
  4 - reserved for future use
  8 - trace LOCKDEV and UNLOCKDEV (x1208)
 16 - trace heap allocates and frees (x1210)
 32 - trace calls to Runtime Entry points
 64 - reserved for future use
128 - reserved for future use
```

Options can be combined in the same run. For example, to trace locks and use Enhanced timer resolution, `SETTRACE T EAMTS10L.286 4617`. To trace heap allocates and frees, and heap scanning in the prolog and epilog, `SETTRACE T EAMTS10L.286 4626`.

The four reserved values above will be reassigned in the future, as needed to activate additional diagnostic features we have not yet thought of.

4.3 SETTRAP

This tool is used to specify what error code(s) should cause a dump. If the CBASIC Runtimes encounters a condition that causes any of the errors you specify when running `SETTRAP`, the Runtimes do not call the current `ON ERROR` routine, but instead cause an abend that results in a dump.

The steps for establishing a trap for a **terminal** application are:

1. Ensure that the application dump flag is set, in "Generic Terminal Configuration" - "Load Definitions" - "General Settings" for the terminal that is experiencing the error.
2. Place the downloaded SB286LT.L86 module in the directory where you link your terminal sales application.
3. Run `LINK86` and then `POSTLINK` on your terminal sales application.
4. Run `SETTRAP` on the executable (for example, `EAMTS10L.286`) and specify `T` for the terminal environment, the load module name, the two-letter ERR that you are interested in, and optionally, a session number. For example:
   ```
   SETTRAP T EAMTS10L.286 FU 52
   ```

The steps for establishing a trap for a **controller** application are:

1. Ensure that the application dump flag is set, in "Controller Configuration" - "Controller Characteristics" for the controller that is experiencing the error.
2. Change the INP file from `SH` to `NOSH` so that the runtimes are embedded into the load module (to be patched).
3. Place the downloaded SB286L.L86 module in the directory where you do your linking.
4. Run `LINK86` and then `POSTLINK` on your controller module.
5. Run `SETTRAP` on the executable and specify `C` for the controller environment, the load module name, the two-letter ERR that you are interested in, and optionally, a session number. For example:
   ```
   SETTRAP C EAMCSMLL.286 RE 08
   ```

From a dump, `APPDUMP` lists the `SETTRAP` codes currently in use. Look for a line like this. In this example, only the SS error is set to trap. "yy" is a placeholder that signifies no code is set for that type of error.

```
      SETTRAP codes are SS, yy, yy, yy.
```

Running `SETTRAP` more than once on the same .286 file will overlay the existing trap codes with the codes from the final `SETTRAP` run. To cancel all trap codes, perform another link.

5.0 History of Runtimes Changes

Here is a pair of tables showing some of the more recent changes made to the Controller and Terminal Runtimes.

To help you determine if you have the desired version of Runtimes in use, you can determine what version you are using by inspecting the `APPDUMP` output from a dump. Look for a line like this:

```
CBASIC runtimes dated 04/20/06, version 54
```

5.1 Terminal Runtimes SB286LT.L86

Terminal Version	Approx. Date	Changes
CBRT054	04/20/2006	Save the SWI disable count in the CBASIC call history trace (instead of BP); also the number of `ON ERROR` occurrences. Allow the application to obtain the SWI disable count.
CBRT055	03/23/2007	Trace calls to `cb_s_enable/disable`.
CBRT056	04/12/2007	Trace first and second (stacked) errors.
CBRT057	10/01/2007	Fix `CBWAIT`.
CBRT058	02/15/2008	Add trace code to count function/subroutine calls. Activated by `SETCALLS`.
CBRT059	10/17/2008	Add function `MILLISECONDTIMER`.
CBRT060	12/08/2008	Fix counter in the last 15 error trace.
CBRT061	02/17/2009	Add Unresumed Error checking. Add Integer tracing.
CBRT062	10/09/2009	Add Runtimes Internal tracing. Call it for Locking and Unlocking.
CBRT063	03/31/2010	Allow Serial port speeds greater than 28800 bps.
CBRT064	04/14/2010	Add tracing for Heap allocates and frees. Add optional checking of the validity of free blocks in the Heap. Function `GETNESTLEVEL` returns I*2 Call Trace nesting level. Subroutine `SCANFREEHEAPBLOCKS` verifies the validity of the free Heap blocks. `alloc_from_freelist` is made `PUBLIC` for easier stack analysis in the event of a heap error.
CBRT065	07/09/2010	Serviceability improvement when review of an Interlisting is required. Added code to CBINIT so that the mainline program always starts at offset 0000 within its selector. Without this change and a related change to the BASIC V3.15 compiler, addresses reported by APPDUMP can be off by zero through 15 bytes as compared to an Interlisting. Note that if you are using ADXMEL0L.L86 in your INP file, you should use a version of that file that is 21,120 bytes in size.
CBRT066	02/28/2011	Serviceability improvement to trace Runtimes entry points.
CBRT067	02/14/2011	Serviceability improvement to trace the two-letter ERR code and the ERRF session number values in the Application Call trace at the time of each error.

5.2 Controller Runtimes SB286L.L86

Controller Version	**Approx. Date**	**Changes**
CBRT636	04/25/2003	Add check for session number to error trap.
CBRT637	07/13/2004	Save/restore nestlevel variable at `ON ERROR GOTO` and `RESUME LABEL.`
CBRT638	03/02/2005	Quadbyte DBCS support.
CBRT639	02/23/2009	Trap unresumed errors. Save Timestamps in the Call trace. Save time of last SVC. Add function `MILLISECONDTIMER`. Trace error count and SWI disable count instead of BP in the call trace.

6.0 SA Debugging Hooks

This section describes some of the places in TGCS Supermarket Application where you can take advantage of built-in hooks for debugging unusual problems. It is intended to include information helpful for programmers to *debug* their SA base or extension code. It is *not* meant to tell you how to turn on existing SA sales or reporting capabilities (such as hidden options), or to help you add new sales or reporting capabilities to SA.

No special SA or Runtimes support is provided, or requires special activation (in most cases). You merely add your own, appropriate code to take advantage of existing capabilities.

6.1 EAMTRACE

If a specific variable (or set of variables) is set to a "bad" value, or an undesirable state change occurs, the result can be a quick abend, or one that is delayed by many function calls or seconds. When the abend is delayed, finding the source-code place where that trashing occurred can be very difficult and time consuming. A new hook was added to the `EAMTRACE` subroutine.

If user exit 69 is enabled, a new subroutine `EAMTRACE.EXIT69` is called each time `EAMTRACE` is called. This happens many times in nearly all main SA functions, and also surrounds nearly every user exit call. Because it is called so frequently, a single test inserted into this subroutine for this "bad" condition has the effect of being inserted in many places. A simple test using most features of SA Base code showed that this subroutine was called more than 230 times during the sale of just one item. Your results will vary depending upon what features, enhancements and hardware are installed, and how they are configured.

Code that you add to `EAMTRACE.EXIT69` can help you to quickly narrow down when, and perhaps where a particular error occurs. You can add whatever checks or modifications you believe are necessary to recover from a potential problem, to capture useful diagnostic information in a file, or to dump the system when it recognizes an "error" state.

One `INTEGER*2` parameter is passed to `EAMTRACE.EXIT69`. It contains two characters, which are *usually* the seventh and sixth characters (note the byte "swapping") of the current source file name. In EAMTS11C, it is 3131H. When a User Exit is about to be called, it is usually 5055H (`UP`, i.e. eamts`UP`c). Note that some features such as Electronic Marketing do not adhere to this naming convention when calling `EAMTRACE`, and some features do not always call `EAMTRACE` everywhere they might be expected to call it. Your added code in `EAMTRACE.EXIT69` should take this into account.

Because `EAMTRACE.EXIT69` is called so frequently, including `ON ERROR` and `ON ASYNC ERROR` context, a large amount of code added to it could introduce performance issues. Minimize your code to the smallest amount necessary to diagnose the immediate problem and capture doc or a dump, or to repair it *in the short term* until you can design and roll out a permanent fix. Furthermore, careless modification of `GLOBAL` variables could cause disastrous results. It is recommended that you only modify *local* variables from within that subroutine (unless you are actually trying to repair a code failure).

This new hook was integrated with SA APAR IO10105 in March 2009. That APAR also provides a stub in `EAMDUMMY` so that a link error does not occur if you do not define your own `EAMTRACE.EXIT69` subroutine.

One good place to insert the code for your new subroutine is in EAMTSUSU.J86. Here is a sample shell. The contrived purpose of the code shown below is to dump on the 1000th call to it.

```
!THIS INCLUDE FILE PROVIDES A PLACE FOR THE USER
!TO DEFINE SUBROUTINES WHICH ARE COMMON TO SEVERAL
!OF HIS USER EXIT FUNCTIONS IN EAMTSUPC.

   %INCLUDE EAMTSESU.J86   ! electronic marketing

  ! User Exit 69 enables calls to this SUB
  SUB EAMTRACE.EXIT69(TRXX) PUBLIC
  INTEGER*2 TRXX
  INTEGER*4 temp1, temp2

  exit69.called = exit69.called +1
  if (exit69.called = 1000) then begin
    exit69.called = 0 ! clear condition
                      ! to prevent recursion

    ! choose one of the 2 methods to dump
    !  (NOT both!)

    ! use the code below with a
    ! controller-terminal, or a discrete terminal.
    ! must use SETTRAP T EAMTSxxL.286 DZ
    temp1 = 0
    temp1 = temp1 / temp1 ! this causes a DZ error

    ! -OR- use this statement for a discrete
    ! terminal.  The SETTRAP is *not* needed
    CALL ADXSERVE(temp2, 1, 0, "")! dump terminal

  endif

  END SUB
```

7.0 License Agreement