AN EXTERNAL DEBUGGING SYSTEM FOR WEAPON SYSTEM PROGRAMS
WRITTEN IN A HIGHER LEVEL LANGUAGE

Jack A. Gaines, Jr.

Naval Surface Weapons Center
Dahlgren, Virginia

Abstract

The operational software for the TRIDENT Fire Control System is written in a procedure-oriented block-structured high level language. To facilitate checkout of this software at the source language level, the TRIDENT On-Line Aid in Debugging Compiler Level (TOADC) System was developed. The system is external, residing in a PDP 11/05 minicomputer, which is connected to the Fire Control Computer by special adapters. It is command-driven, providing both interactive and batch operation.

The interface between TOADC and the subject operational program is a file of information, known as the Schema, which provides a mapping of the listed form of the program to the corresponding executable form.

Introduction

The operational software for the TRIDENT Fire Control System is written in a high level language known as THLL (for TRIDENT Higher Level Language) [1]. THLL is an ALGOL-like, procedure-oriented, block-structured language with special features for data structure manipulation and fire control operations. At the time THLL was designated to be used, the only debugging facilities available for the TRIDENT Digital Control Computer (TDCC) were at the assembly/machine language level. Due to the capabilities of THLL (e.g., shared variables defined dynamically on a runtime stack), these facilities were deemed inadequate for the tasks of debug and checkout of the operational software. Thus, a new debug facility, the TRIDENT On-Line Aid in Debugging Compiler Level (TOADC) System, was developed.

TOADC is an external, command-driven system with facilities to allow debug to be performed in terms of THLL statements and data constructs rather than in terms of their machine level representations. The command structure is designed to support both interactive and batch operation of the system. No modification of the compiler-generated code of the program being debugged is required to use TOADC.

TOADC Operating Environment

The system in which TOADC resides is known as the Software Development System. This system consists of a Digital Equipment Corporation (DEC) PDP 11/05 minicomputer with a DECwriter terminal, CRT display, card reader, line printer, two disk units, two magnetic tape units, and special adapters. The adapters connect the 11/05 to the two components of the TDCC, the Basic Processor (BP) and the Data Communications Processor (DCP). They allow the 11/05 to control TDCC operation and provide access to all TDCC registers and memory.

TOADC operates under BOSS, a modified version of the DEC-supplied operating system DOS/BATC V09-20C. The modifications provide a software interface to the BP and DCP adapters.

As stated above, the SDS has two disk units available. For TOADC, one of these units is designated as the System Unit. On it reside DOS, TOADC, DEC-supplied utilities, and other SDS support programs. The other unit is designated as the User Unit. It is used for storage of all the user data files which form the data base needed by TOADC for debugging a given subject program. Individual users are allocated private data disk packs to add a measure of protection for these files.

The Program Schema

One of the goals established for the TOADC system was that the compiled code of the subject program should not require modification, either during compilation or testing, to perform the tasks of debug and checkout. To accomplish this goal, the concept of a Program Schema (as described by Mann [2]) is utilized. The schema is a file of information mapping the THLL statements and data constructs of a program into their corresponding machine-level representations. The THLL compiler and the TASS program (the relocating linker/loader which combines separate THLL compilation units into executable load modules) produce this file.

U.S. Government work not protected by U.S. copyright.
The schema for a program is generated in an incremental fashion. The THLL compiler produces schema data for each unit that is compiled, where a unit may contain data declarations and/or one or more procedures. This schema data consists of the following items:

1. A table of procedure and block structure information,
2. Tables of symbolic data (variables, formal parameters, etc.) information, arranged according to the blocks in which the data is defined, and
3. A table of statement structure information.

These tables are appended to the output form of the unit produced by the compiler. The information contained in the tables is strictly relative to that unit; e.g., addresses of variables are saved as offsets relative to the compile unit rather than as absolute addresses.

Once all of the units comprising a program are compiled, they are formed into an executable load module by a three-step process utilizing TASS. First, the compile units are relocated in a process known as binding. Information detailing this relocation is added to the procedure/block structure table for each compile unit, and the schema data for all of the relocated compile units is merged together into a single entity, which is appended to the relocated output of the binding process. Within this entity, the schema data is still arranged by compile units. Next, the relocated compile units are merged together (linked) into a logical segment structure to produce a skeletal load module. The linking process adds a set of tables to the schema detailing the segment structure and the organization of the compile units within the segments. Lastly, the skeletal load module is formed (loaded) into the overlay structure comprising the executable load module. During loading, a table of information about the program (entry point, number of overlays, etc.) and the arrangement of the logical segments in the overlays is added to the schema, thereby completing it.

The completed schema is appended to the executable load module, eliminating the possibility of a mismatch occurring between a program and its schema. This single entity is then transported to the SDS on a magnetic tape. The load module is transferred to the TRIDENT Magnetic Disk File (MDF) for execution. The schema is loaded onto the SDS User Disk as a random access file, to which a directory of all compile units contained in the schema is added. This directory enables quick TOADC access to the data for any requested compile unit. The total program generation process is shown in Figure 1.

TOADC Capabilities - Overview

TOADC provides the user with a full range of capabilities allowing debug to proceed at the THLL source language level and, when necessary, at the assembly/machine language level. The user can examine and modify both the data and the compiler-generated code of the subject program. He can execute the subject program in a controlled manner, stopping as necessary to initialize and/or dump data values automatically. He can invoke predefined files of checkout procedures. Within these files, and within batch debug jobs, the flow of the checkout session can be determined based on conditions within the subject program.

To invoke a particular capability of TOADC, the user enters the appropriate command. If no further information is required, the system executes the command and then awaits entry of the next command by the user. If further information is required, the system prompts the user for it. When this dialogue of prompts and user responses is complete, the system executes the command.
Commands which prompt the user for further information may be one-time or cyclic, depending on the action of TOADC when the prompting sequence and command execution are complete. For a one-time command, TOADC simply awaits the entry of another command by the user. For a cyclic command, TOADC reinitiates the prompting sequence for the command, thus allowing multiple operations to be performed with a single command invocation. To exit from a cyclic command, the user must enter a special escape entry as a prompt response. This escape entry will cause TOADC to await the entry of another command by the user. The escape entry can also be used to escape from the prompting sequence of any prompting command, thereby aborting the execution of the command.

Addressing Capabilities

TOADC has provisions for three methods of addressing TDCC memory: symbolic, virtual, and physical. Symbolic addressing is in terms of the THLL constructs within the compile units comprising the subject program. Such addresses are divided into two categories. A symbolic program address references the executable statement structure of a compile unit. It consists of a compile unit name and a line number, as shown on the compiler-generated listing of the compile unit. It must refer to the beginning of a THLL statement. When there is more than one statement on a single line, a sequence number must be included with the line number to reference the individual statements. A symbolic data address references a data item defined within a compile unit. It consists of a compile unit name, the line number on which the variable is defined, and the variable specification (name and, if needed, subscripts). The line number is required due to the block orientation of THLL, in which the same variable name may be defined in more than one block of a compile unit, with each definition representing a distinct variable.

Virtual addressing is in terms of the base/protection register settings in the TDCC. A virtual address consists of the designation of a base register number and a displacement relative to that base register. Since the TDCC is a two-state machine, a virtual address also requires the designation of the desired state to determine which set of base registers is being referenced. This state designation may be entered as a part of the virtual address (implicit state designation) or as a separate entry (explicit state designation).

Physical addressing is in terms of the actual absolute memory of the TDCC. A physical address consists simply of the hardware address assigned to the desired memory location. This type of addressing is particularly useful for addressing the TDCC processor registers, which are assigned to the first 256 locations in TDCC memory.

Data Manipulation Capabilities

THLL requires that each variable be declared to be one of the following types:

- Half integer (legal for arrays only),
- Integer,
- Double integer,
- Real,
- Pointer,
- Alpha.

This type, together with data detailing the memory allocation and structure of the variable (e.g., array), is included in the entry for the variable in the appropriate schema symbol table. TOADC uses this information to determine the format to be used when outputting the contents of a variable for user examination.

TOADC also allows the user to override the typing information contained in the schema. This is necessitated by a feature of THLL known as components. Components allow the programmer to access fields in words under different data types than the original declaration for the words, thus allowing the definition of complex data structures. However, due to the manner in which the compiler processes components, no information concerning them is available for insertion into the schema. To access these data structures in a meaningful way with TOADC, the user must override the type declaration from the THLL program. This override format, for output, is included as a part of the symbolic data address of the variable being accessed. For input, the format is implicit in the data item. The format overrides available are as follows:

- Half decimal,
- Half hexadecimal,
- Integer decimal,
- Integer hexadecimal,
- Double decimal,
- Double hexadecimal,
- Real,
- Pointer (hexadecimal),
- Alpha,
- Binary,
- Octal.
These formats are also allowed when working with virtual and physical addresses, which, since they have no schema typing data, are defaulted to integer hexadecimal format.

The user has the option of examining variable values interactively, one value at a time, or of having the system dump or initialize the values in groups. If interactive examination is chosen, a new value may be inserted into the variable after the current value is output. If group operation is chosen, the user must first define one or more lists of addresses in which he is interested, where each list constitutes a data group. To facilitate data group definition with respect to subscripted items (e.g., array elements) and virtual and physical addresses, the user may specify the starting address and a count of the number of items desired. This count is included as a part of the address entered. After the data groups are defined, they may be used whenever desired by supplying the group names to the appropriate command. For a dump, the data items are either written on magnetic tape or printed in the order specified by the data group. For initialization, data values are read from cards or magnetic tape and stored in the addresses specified by the data group.

**Dynamic Capabilities**

Several of the capabilities provided by TOADC deal with controlling the execution of the subject program. Initially, the user must identify the subject program (and its schema) to TOADC. Since the TDCC operates in a multitasking environment, TOADC must then wait (if necessary) until the subject program is loaded into TDCC memory and readyed for execution.

Once the program is loaded into TDCC memory, the user has several options. The program can be run and halted on user command. Events, in the form of breakpoints and the mousetrap, can be set to halt TDCC execution at selected points within the subject program. Breakpoints are set on THLL statements or on program variables to halt TDCC execution when the statement is reached or the variable is accessed. The mousetrap is a limit checking device set on a program variable. Whenever a value is stored into the variable, the mousetrap compares that value to the lower and upper limits set by the user. If the value is within the limits, or outside of them, depending on the option chosen by the user, the mousetrap halts the BP. Breakpoints and the mousetrap are mechanized in TDCC hardware.

When setting an event, the user can link a set of actions to it for TOADC to execute when that event occurs. This list can contain several functions. It can request the dumping of values as specified by data groups. It can also request that variables in data groups be initialized with values from cards or magnetic tape. It can request that new breakpoints be set or current breakpoints be modified or deleted. Lastly, it can request that the mousetrap be set or that the current setting be modified or deleted.

After setting one or more events, the user can run the subject program. TOADC monitors TDCC execution, awaiting occurrence of the set events. When an event occurs which has an actions list associated with it, the actions in that list are performed by TOADC. Upon completion of the list, TOADC automatically restarts the TDCC, provided that such a restart was requested when the event was set. If automatic restart was not requested, or if no actions list was associated with the event, TOADC simply awaits further commands from the user.

Should problems be encountered in a program for which the user needs more detailed information than can be obtained by setting events and executing the program, TOADC provides the capability to step the program and to trace its execution. Stepping can be performed at two levels. Statement stepping proceeds from the current THLL statement to the beginning of the next statement in the execution sequence. The user sees these statements in terms of line/sequence numbers within the current compile unit and relevant variable values as defined by the statement position in the compile unit block structure. Instruction stepping consists of executing individual machine language instructions in the TDCC. During instruction stepping, TOADC displays the contents of relevant TDCC processor registers.

**Tracing of a THLL program consists of executing the program and reporting the sequence of statements encountered. Procedure calls and returns, and any associated parameters, are also reported.**

A further capability provided by TOADC is procedure traceback. When the TDCC is halted within a THLL procedure (e.g., by a breakpoint occurrence), this capability allows the user to examine the sequence of procedure calls which led to that procedure.

**Debug Procedure Invocation Capabilities**

In the course of debugging a subject program, certain sequences of TOADC commands and prompt responses are repeated many times with little or no change (e.g., subject program starting procedures, fault halt dumps). TOADC provides the user with a way to avoid repeated entry of these sequences. External to TOADC (e.g., via the DEC-supplied Text Editor program), files may be defined containing the command/prompt response sequences. These are known as execute files. During debug, the user may command TOADC to invoke such a file. The system then proceeds to take the commands and prompt responses from the file rather than from the user. When the file is completed, TOADC returns control to the user. If requested, TOADC will repeat the execution of the file n times, where n is set by the user, before returning control.

For command/prompt sequences in which some of the entries are variable, TOADC provides a textual substitution type of parameter passage. When defining the file to be executed, the user may insert special commands of the form /m, where m is in the range 1 to 9, indicating that the command/prompt response
in that position will be entered as a parameter when the file is invoked. The user may also insert a default text with the special command, to be used if no corresponding parameter text is defined. To execute the file, the user invokes it and informs TOADC that textual substitution parameters are to be defined. The user then enters, in response to a TOADC prompting sequence, the parameter texts (commands/prompts) along with an indication of the special commands in the file for which they are to be substituted. Upon completion of this prompting sequence, TOADC executes the requested file. For each /m special command encountered, TOADC substitutes the corresponding parameter text. If the user has not defined a corresponding parameter text, TOADC uses the default text included in the file.

Invocation of files is available during both interactive and batch operation, and the invocations may be nested. Due to limitations imposed by BOSS, recursive invocations cannot be supported.

Conditional Capabilities

While operating TOADC interactively, the user can observe conditions within the subject program (e.g., paths taken, variable values) and base subsequent debug actions on these conditions. During batch operation, and when operating from an execute file, similar condition-dependent debug is available through use of the conditional structures provided by TOADC. These conditional structures are of two forms: alternate-execution-path structures and looping structures.

Alternate-execution-path structures provide an If-Then-Else capability. The user defines the set of subject program conditions on which TOADC is to base the execution path choice. These conditions may include breakpoint occurrences (for determination of subject program paths taken), mousetrap occurrence, and variable values. If the conditions are met, the first group of commands in the structure are executed; otherwise, control passes to the second group of commands in the structure.

Looping structures provide a While-Do capability. The user defines a set of subject program conditions of the same type used with the alternate path structures. If these conditions are met, the TOADC commands contained within the loop body are executed. When the loop body is completed, control returns to the conditions, which are reevaluated. If the conditions are not met, the commands within the loop body are skipped, with control passing to the command following the loop body.

Both forms of conditional structures can be nested and intermixed, allowing the user to control the flow of noninteractive debug completely. In addition, the looping structures are also available during interactive operation to permit the user to carry out repetitive, condition-based command sequences with relative ease. The alternate-execution-path structures are not available during straight interactive operation except within looping structures and execute files. The interactive user can check the conditions and make the necessary decisions directly.

Conclusion

In general, software debug and checkout for weapon systems must be carried out at the assembly/machine language level, even when the software is written in a higher level language. In the TRIDENT Fire Control System, this is not true. TOADC provides a full range of debug capabilities to the THLL programmer, allowing debug to proceed at the source language level and shifting many of the detailed burdens (e.g., dynamic address calculation) from the programmer to the debug system.

THLL programmers have an additional debug resource. An emulation of the BP and several of its peripherals has been generated on a Nanodata QM-1. To debug programs on this emulation, the Dynamic Debug for TRIDENT (DDT) System was developed. DDT is essentially a QM-1 based version of TOADC, sharing the same command/prompt structure and most of the same services. TOADC and DDT together provide TRIDENT operational programmers with a wide range of resources for debug and checkout.

References