iASL: ACPI Source Language Optimizing Compiler and Disassembler

User Guide

iASL Overview and Compiler Operation

Revision 4.05

January 12, 2011

Contents

1 Introduction

The iASL compiler/disassembler is a fully-featured translator for the ACPI Source Language (ASL) and ACPI binary data tables. As part of the Intel ACPI Component Architecture, the Intel ASL compiler implements translation for the ACPI Source Language (ASL) to the ACPI Machine Language (AML). The disassembler feature will disassemble compiled AML code back to (near-original) ASL source code.

The major features of the iASL compiler include:

- Full support for the ACPI 4.0a Specification including ASL grammar elements and operators.
- Extensive compiler syntax and semantic error checking, especially in the area of control methods. This reduces the number of errors that are not discovered until the AML code is actually interpreted (i.e., the compile-time error checking reduces the number of run-time errors.)
- Multiple types of output files. Besides binary ACPI tables, output options include formatted listing files with intermixed source, several types of AML files, and error messages.
- Automatic detection and compilation of either ASL source code or ACPI data table source code.
- Portable code (ANSI C) and source code availability allows the compiler to be easily ported and run on multiple execution platforms.
- Support for integration with the Microsoft Visual C++ (or similar) development environments.
- Disassembly of all ACPI tables, including tables that contain AML (DSDT, SSDT) as well as ACPI “data” tables such as the FADT, MADT, SRAT, etc.
- Support for compilation of non-AML data tables such as the FADT, MADT, SRAT, etc.

1.1 Document Structure

This document consists of these major sections:

Introduction**:** Contains a brief overview of the iASL compiler/disassembler, document structure, related reference documents, and definition of terms used throughout the document

Compile/Disassembler Overview: Compiler subsystems, inputs, outputs, and supported system environments.

ASL-AML Subsystem: Describes the ASL compiler and the AML disassembler.

ACPI Data Table Subsystem: Describes the Data Table compiler and the Data Table disassembler.

Compiler/Disassembler Operation: Guide for compiler options and general operation, including output examples.

Generating iASL from Source Code: Instructions for building the iASL compiler from the open-source package.

1.2 Reference Documents

ACPI documents are available at:
http://www.acpi.info

Advanced Configuration and Power Interface Specification, Revision 1.0, December 1, 1996
Advanced Configuration and Power Interface Specification, Revision 1.0a, July 1, 1998
Advanced Configuration and Power Interface Specification, Revision 1.0b, February 8, 1999
Advanced Configuration and Power Interface Specification, Revision 2.0, July 27, 2000
Advanced Configuration and Power Interface Specification, Revision 2.0a, March 32, 2002
Advanced Configuration and Power Interface Specification, Revision 2.0b, October 11, 2002
Advanced Configuration and Power Interface Specification, Revision 2.0c, August 23, 2003
Advanced Configuration and Power Interface Specification, Revision 3.0, September 2, 2004
Advanced Configuration and Power Interface Specification, Revision 3.0a, December 30, 2005
Advanced Configuration and Power Interface Specification, Revision 3.0b, October 10, 2006
Advanced Configuration and Power Interface Specification, Revision 4.0, June 16, 2009
Advanced Configuration and Power Interface Specification, Revision 4.0a, April 5, 2010

ACPICA documents are available at:
http://www.acpica.org/documentation/

ACPI Component Architecture User Guide and Programmer Reference
iASL: ACPI Source Language Optimizing Compiler and Disassembler User Guide

ACPICA and iASL source code is available at:
http://www.acpica.org/downloads/

iASL Windows binaries are available at:
http://www.acpica.org/downloads/binary_tools.php

1.3 Definition of Terms

ACPI: Advanced Configuration and Power Interface. An open standard for device configuration and power management.

ACPICA: ACPI Component Architecture. An open-source implementation of ACPI that is hosted on many different operating systems.

ACPI Data Table: Any ACPI table that does not contain AML byte code but is instead simply a structure of static packed binary data. In practice, any ACPI table other than DSDTs or SSDTs.

ACPI Table: Generic reference to any of the ACPI-related tables (both AML and Data Tables) that are presented by the BIOS for consumption by the host operating system.

AML: ACPI Machine Language. A byte code language to be executed by an ACPI/AML interpreter within the host operating system. Created by translation of ASL code via an ASL compiler. Defined by the ACPI specification.

ASL: ACPI Source Language. A higher level language that corresponds to the low level AML byte code language. ASL source code is translated into AML byte code by an ASL compiler. Defined by the ACPI specification

Binary ACPI Table: An ACPI table that contains either raw AML byte code, or a packed ACPI Data Table

Data Table Language: A simple language developed to describe the individual fields within an ACPI Data Table. It is used by both the compiler and disassembler portions of the iASL Data Table Subsystem.

Disassembler: In the ACPI context, a tool that will either convert AML byte code back to the original ASL code, or will convert an ACPI Data Table into a format that is human-readable.

Hex Table: A table containing data that is in a format suitable for translation via an Assembler, C compiler, or ASL compiler.

2 Compiler/Disassembler Overview

The iASL compiler/disassembler consists of several distinct subsystems, as described below:

- An ASL-to-AML compiler that translates ASL code (ACPI Source Language) to AML byte code (ACPI Machine Language).
- An ACPI Data Table compiler that translates Data Table definitions to binary ACPI tables. An ACPI Data Table is any ACPI table that contains only data, not AML byte code. Examples include the FADT, MADT, SRAT, etc.
- An AML-to-ASL disassembler that translates compiled AML byte code back to the (nearly) original ASL source code. This disassembler is used on tables like the DSDT and SSDT.
- An ACPI Data Table disassembler that formats binary ACPI data tables into a readable format. The output of this disassembler can be compiled with the ACPI data table compiler.
- An ACPI table template generator that will emit examples of all known ACPI tables, in a format similar to the output of the data table disassembler. The output files from the template generator are intended to be used as the basis or starting point for the development of actual ACPI tables.

2.1 Supported Execution Environments

iASL runs on multiple platforms as a 32-bit application. Generation and operation as a 64-bit operation has not been tested and is not supported at this time.

Portable code – requires only ANSI C and a compiler generation package such as Bison/Flex or Yacc/Lex.

Error and warning messages are compatible with Microsoft Visual C++, allowing for integration of the compiler into the development environment to simplify project building and debug.

The iASL source code is distributed with the compiler binaries under the ACPICA source license.

2.2 ASL Compiler

2.2.1 Input Files

Existing ACPI ASL source files are fully supported. Enhanced compiler error checking will often uncover unknown problems in these files.

All ACPI 4.0a ASL additions are supported. The compiler fully supports ACPI 4.0a.

2.2.2 Output File Options

- AML binary output file
- AML code in C source code form for inclusion into a BIOS project

- AML code in x86 assembly code form for inclusion into a BIOS project
- AML Hex Table output file in either C, ASL, or x86 assembly code as a table initialization statement.
- Listing file with source file line number, source statements, and intermixed generated AML code. Include files named in the original source ASL file are expanded within the listing file
- Namespace output file — shows the ACPI namespace that corresponds to the input ASL file (and all include files.)
- Debug parse trace output file — gives a trace of the parser and namespace during the compile. Used to debug problems in the compiler, or to help add new compiler features.

2.3 AML Disassembler

The AML Disassembler has the capability of reverse translating any binary AML table back to nearly the original ASL code. These are typically DSDTs and SSDTs.

2.3.1 Input Files

The AML Disassembler accepts binary ACPI tables that contain valid AML code. These tables are the DSDT and any SSDTs.

These files may be obtained via the acpidump/acpixtract utilities, or some other host-specific tools.

2.3.2 Output

The output is disassembled (or de-compiled) ASL code. The file extension used for these output files is .DSL, meaning “disassembled ASL”. As opposed to original ASL source code files which typically have the extension .ASL.

2.4 Data Table Compiler

The Data Table compiler is used to compile the “non-ASL/AML” ACPI tables such as the FADT, MADT, SRAT, etc. These tables are not compiled to AML byte code, but are compiled to simple binary data, usually with the standard ACPI table header (signature, length, checksum, etc.)

The intent of the Data Table Compiler is to simplify the generation of the many non-ASL ACPI data tables and to make the generation process less error-prone. The Data Table Compiler knows the required format for each recognized ACPI table, as well as the exact size and allowable values for each field within the tables.

2.4.1 Input Files

The Data Table compiler accepts as input files that are in the same or simplified format as the files emitted by the data table disassembler. An existing ACPI binary data table may be disassembled, modified, and then recompiled.

Also, the ACPI table template generator may be used to generate template ACPI data tables that can in turn be used for the basis for additional table development. This would be the preferred starting point for ACPI table development, since the ACPI table templates contain a valid example of each table header, table section, and table sub-table as applicable.

2.4.2 Output

- Binary output file
- Hex Table output file in either C, ASL, or x86 assembly code as a table initialization statement for inclusion into a BIOS project.

2.5 Data Table Disassembler

This second part of the disassembler package will extract all data from a binary ACPI "data table" and format it into human readable form. The format of this output is compatible with the Data Table Compiler, meaning that such ACPI tables may be easily disassembled, modified, and recompiled.

2.5.1 Input Files

The Data Table Disassembler accepts binary ACPI tables that do not contain AML code. These tables include the FADT, MADT, SRAT, etc.

2.5.2 Output

The output is a disassembled and formatted ACPI table in human-readable format. The file extension used for these files is also .DSL, for consistency with the AML disassembler.

2.6 Template Generator

The iASL Template Generator can be used to create ACPI tables from templates that are stored within the iASL image. These templates can be used as a starting point for the development of any ACPI table known to the compiler.

3 ASL-AML Subsystem

This subsystem consists of tools to compile ASL source code to AML byte code, and disassemble AML byte code back to the original ASL code.

3.1 ASL Compiler

The iASL compiler fully supports ACPI 4.0a. The ASL and AML languages are defined within the ACPI specification.

3.1.1 Compiler Analysis Phases

3.1.1.1 General ASL Syntax Analysis

Enhanced ASL syntax checking. Multiple errors and warnings are reported in one compile – the compiler recovers to the next ASL statement upon detection of a syntax error.

Constants larger than the target data size are flagged as errors. For example, if the target data type is a BYTE, the compiler will reject any constants larger than 0xFF (255). The same error checking is performed for WORD and DWORD constants.

3.1.1.2 General Semantic Analysis

All named references to objects are checked for validity. All names (both full ACPI Namepaths and 4-character Namesegs) must refer to valid declared objects.

All Fields created within Operation Regions and Buffers are checked for out-of-bounds offset and length. The minimum access width specified for the field is used when performing this check to ensure that the field can be properly accessed.

3.1.1.3 Control Method Semantic Analysis

Method local variables are checked for initialization before use. All locals (LOCAL0 – LOCAL7) must be initialized before use. This prevents fatal run-time errors for uninitialized ASL arguments.

Method arguments are checked for validity. For example, a control method defined with 1 argument can't use ARG4. Again, this prevents fatal run-time errors for uninitialized ASL arguments.

Control method execution paths are analyzed to determine if all return statements are of the same type — to ensure that either all return statements return a value, or all do not. This includes an analysis to determine if execution can possibly fall through to the default implicit return (which does not return a value) at the end of the method. A warning is issued if some method control paths return a value and others do not

3.1.1.4 Control Method Invocation Analysis

All control method invocations (method calls) are checked for the correct number of arguments in all cases, regardless of whether the method is invoked with argument parentheses or not (e.g. both ABCD() and ABCD). Prevents run-time errors caused by non-existent arguments.

All control methods and invocations are checked to ensure that if a return value is expected and used by the method caller, the target method actually returns a value.

3.1.1.5 Predefined ACPI Names

For all ACPI reserved control methods (such as _STA, _TMP, etc.), both the number of arguments and return types (whether the method must return a value or not) are checked. This prevents missing operand run-time errors that may not be detected until after the product is shipped.

Predefined names that are defined with arguments or return no value must be implemented as control methods and are flagged if they are not. Predefined names that may be implemented as static objects via the ASL Name() operator are typechecked.

Reserved names (all names that begin with an underscore are reserved) that are not currently defined by ACPI are flagged with a warning.

3.1.1.6 Resource Descriptors

Validation of values for Resource Descriptors is performed wherever possible.

Address Descriptors: Values for AddressMin, AddressMax, Length, and Granularity are validated:

AddressMax must be greater than or equal to AddressMin

Length must be less than or equal to (Max-Min+1)

If Granularity is non-zero, it must be a power-of-two minus one.

The IsMinFixed and IsMaxFixed parameters are validated against the values given for the AddressMin, AddressMax, Length, and Granularity. This implements the rules given in Table 6-40 of the ACPI 4.0a specification.

3.1.2 Compiler Optimizations

The compiler implements several optimizations whose primary intent is to reduce the size of the resulting AML output.

3.1.2.1 Named References

Namepaths within the ASL can often be optimized to shorter strings than specified by the ASL programmer. For example, a full pathname can be optimized to a single 4-character ACPI name if the final name in the path is within the local scope or is along the upward search path to the root from the local scope. In addition, the carat (^) operator can often be used to optimize Namepaths.

3.1.2.2 Integers

Certain integers can be optimized to single-byte AML opcodes. These are: 0, 1, and -1. The opcodes used are Zero, One, and Ones. All other integers are described in AML code using the smallest representation necessary – either Byte, Word, DWord, or QWord.

3.1.2.3 Constant Folding

All expressions that can be evaluated at compile-time rather than run time are executed and reduced to the simplified value. The ASL operators that are supported in this manner are the Type3, Type4, and Type5 operators defined in the ACPI specification.

The iASL compiler contains the ACPICA AML interpreter which is used to evaluate these expressions at compile time.

3.2 ASL-to-AML Disassembler

The AML disassembler is used to regenerate the original ASL code from a binary ACPI AML table. Tables that contain AML are typically the DSDT and any SSDTs.

The disassembler is invoked by using the –d option of iASL.

Because the AML contains all of the original symbols from the ASL, the AML byte code of a binary ACPI table can be disassembled back to nearly the original ASL code with only a few caveats.

3.2.1 Multiple Table Disassembly

There is a known difficulty in disassembling control method invocations for methods that are external to the table being disassembled. This is because there is often insufficient information within the AML to properly disassemble these method invocations.

Therefore, whenever possible, all DSDTs and SSDTs for a given machine should be disassembled together using the –da or –e option. If all SSDTs are included this way, the necessary information will be available to fully and correctly disassemble the target table.

For example, to disassemble the DSDT on a machine with multiple SSDTs:

```
$ iasl -essdt1.dat,ssdt2.dat,ssdt3.dat -d dsdt.dat

Intel ACPI Component Architecture
AML Disassembler version 20100528 [May 28 2010]
Copyright (c) 2000 - 2010 Intel Corporation
Supports ACPI Specification Revision 4.0a

Loading Acpi table from file DSDT.dat
Acpi table [DSDT] successfully installed and loaded
Loading Acpi table from file ssdt1.dat
Acpi table [SSDT] successfully installed and loaded
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Loading Acpi table from file ssdt2.dat
Acpi table [SSDT] successfully installed and loaded
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Loading Acpi table from file ssdt3.dat
Acpi table [SSDT] successfully installed and loaded
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)
.................................................................................
Parsing completed
Disassembly completed, written to "DSDT.dsl"
```

3.2.2 External Declarations

During disassembly, any ACPI names that cannot be found or resolved within the table under disassembly are added to a list of externals that are emitted at the start of the table definition block, as shown below:

```
DefinitionBlock ("DSDT.aml", "DSDT", 1, "INTEL ", "EXAMPLE", 0x06040000)
{
    External (Z003)
    External (\_SB_.PCI0.LNKH)
```

If the object type that is associated with the name can be resolved during the disassembly, this type is emitted with the extenal statement also:

```
    External (PETE, IntObj)
    External (HDOS, MethodObj)    // 0 Arguments
    External (ECST, MethodObj)    // 1 Arguments
    External (PSEN, FieldUnitObj)
    External (C7EN, FieldUnitObj)
    External (\_PR_.CPU1, DeviceObj)
    External (\_PR_.CPU0, DeviceObj)
```

4 ACPI Data Table Subsystem

This subsystem consists of tools to compile ACPI Data Tables such as the FADT, MADT, SRAT, etc., to binary ACPI tables, and to disassemble binary ACPI data tables to formatted and structured tables in the data table language.

4.1 Data Table Compiler

The iASL Data Table Compiler is intended to compile ACPI data tables (FADT, MADT, etc) to binary data, to be integrated into a BIOS project.

Data Tables are described in a simple language that is directly compatible with the output of the data table disassembler. The two goals for this language are simplicity and compatibility with the disassembler.

Data Table input files are automatically detected and differentiated from ASL files, therefore no special iASL option is required to invoke the data table compiler.

The default output is a binary ACPI data table. Use one of the iASL options **–ta**, **–tc**, or **–ts**, in order to create the binary output in an ASCII hex table that is suitable for direct inclusion into a BIOS project.

On some host operating systems, the iASL data table disassembler and compiler may be used to disassemble a data table, modify it, then recompile it to a binary file that can be used to override the original table. This override support depends upon features supported by the host operating system. This feature would be useful, for example, to repair invalid or incorrect values in an important table such as the FADT.

4.1.1 Input Format

The format of the input file is a series of fields, each of which represents a field in the target ACPI table. Each field is comprised of a field name and a field value, separated by a colon. The fields must appear in the exact order in which they are defined for the target ACPI table.

```
<FieldName> : <Field Value>
<FieldName> : <Field Value>
    .
    .
    .
<FieldName> : <Field Value>
```

Both slash-asterisk and slash-slash comments are supported. Blank lines are ignored.

The language itself is defined in the next section. The Field Names (AcpiTableFieldName) that are available for any given ACPI table can be obtained from the template file generated by the iASL Template Generator:

```
iASL –T <ACPI Table Signature>    // Obtain one template
iASL –T all                       // Obtain templates for all ACPI tables
```

4.1.1.1 Ignored Fields/Comments

Comments can be either traditional /* .. */ style or // style.

Fields that are ignored (and are essentially comments) are fields surrounded by brackets – [..] or angle brackets < …> . This allows automatic compatibility with the output of the AML disassembler.

4.1.2 Data Table Definition Language

```
// Root Term

DataTable :=
     FieldList

// List Terms

FieldList :=
     Field | <Field FieldList>

Field :=
     <AcpiTableFieldName ‘:’ FieldValue OptionalFieldComment>

FieldValue =
     Integer | Flags | String | Buffer

AcpiTableFieldName :=
     [Valid name as defined by iASL data table disassembler]

OptionalFieldComment :=
     Nothing | ‘(‘ AsciiCharList ‘)’

// Data Terms

Integer :=
     ByteConst | WordConst | Const24 | DWordConst | Const56 | QWordConst
Flags :=
     OneBit | TwoBits
String :=
     ‘”’ AsciiCharList ‘”’
Buffer :=
     ByteConstList

// Numeric and String Value Terms

ByteConst :=
     0x00-0xFF
WordConst :=
     0x0000 - 0xFFFF
Const24 :=
     0x000000 - 0xFFFFFF
DwordConst :=
     0x00000000 - 0xFFFFFFFF
Const56 :=
     0x00000000000000 - 0xFFFFFFFFFFFFFF
QWordConst :-
     0x0000000000000000 - 0xFFFFFFFFFFFFFFFF

OneBit :=
     0 - 1
TwoBits :=
     0 - 3

ByteConstList :=
     ByteConst | <Byte Const ‘,’ ByteConstList>

AsciiCharList :=
     Nothing | PrintableAsciiChar | <PrintableAsciiChar AsciiCharList>
PrintableAsciiChar :=
     0x20 – 0x7E
```

4.1.3 Input Example

Input is similar to the output of the Data Table disassembler. The example below shows a portion of input describing a FADT.

```
/*
 * Intel ACPI Component Architecture
 * iASL Compiler/Disassembler version 20100528
 *
 * Template for [FACP] ACPI Table
 * Format: [ByteLength]  FieldName : HexFieldValue
 */

[004]                        Signature : "FACP" // Fixed ACPI Description Table
[004]                     Table Length : 000000F4
[001]                         Revision : 04
[001]                         Checksum : 4E
[006]                           Oem ID : "INTEL "
[008]                     Oem Table ID : "TEMPLATE"
[004]                     Oem Revision : 00000000
[004]                  Asl Compiler ID : "INTL"
[004]            Asl Compiler Revision : 20100528

[004]                     FACS Address : 00000001
```

Each valid, non-comment line in the input file represents a field within the target ACPI table. The value in brackets (e.g., “[004]”) is the required length (in bytes) of the field described on the line. It is essentially a comment and is not required; this value is created by the iASL template generator for reference purposes only.

4.1.4 Data Types for User-Entered Fields

The following data types are supported:

Integers: All integers in ACPI are unsigned. Four major types of unsigned integers are supported by the compiler: *Bytes*, *Words*, *DWords* and *QWords.* In addition, for special cases, there are some odd sized integers such as 24-bit and 56-bit. The actual required width of an integer is defined by the ACPI table. If an integer is specified that is numerically larger than the width of the target field within the input source, an error is issued by the compiler. Integers are expected by the data table compiler to be entered in hexadecimal with no “hex” prefix.

Examples:

```
[001]                         Revision : 04                  // Byte (8-bit)
[002]                       C2 Latency : 0000                // Word (16-bit)
[004]                     DSDT Address : 00000001            // DWord (32-bit)
[008]                          Address : 0000000000000001    // QWord (64-bit)
```

Length of non-power-of-two examples:

```
[003]                         Reserved : 000000              // 24 bits
[007]                     Capabilities : 00000000000000      // 56 bits
```

Flags: Many ACPI tables contain flag fields. For these fields, only the individual flag bits need to be specified to the compiler. The individual bits are aggregated into a single integer of the proper size by the compiler.

Examples:

```
[002]            Flags (decoded below) : 0005
                              Polarity : 1
                          Trigger Mode : 1
```

In this example, only the Polarity and Trigger Mode fields need to be specified to the compiler (as either zero or one). The compiler then creates the final 16-bit Flags field for the ACPI table.

Strings: Strings must always be surrounded by quotes. The actual string that is generated by the compiler may or may not be null-terminated, depending on the table definition in the ACPI specification. For example, the *OEM ID* and *OEM Table ID* in the common ACPI table header (shown above) are fixed at six and eight characters, respectively. They are not necessarily null terminated. Most other strings, however, are of variable-length and are automatically null terminated by the compiler. If a string is specified that is too long for a fixed-length string field, an error is issued. String lengths are specified in the definition for each relevant ACPI table.

Escape sequences within a quoted string are not allowed. The backslash character '\' refers to the root of the ACPI namespace.

Examples:

```
[008]                        Oem Table ID : "TEMPLATE"        // Fixed length
[006]                Processor UID String : "\CPU0"           // Variable length
```

Buffers: A buffer is typically used whenever the required binary data is larger than a QWord, or the data does not fit exactly into one of the standard integer widths. Examples include UUIDs and byte data defined by the SLIT table.

Examples:

```
// SLIT entry

[032]                Locality   0 : 0A,10,16,17,18,19,1A,1B,1C,1D,1E,1F,20,21,22,23,
                                  : 24,25,26,27,28,29,2A,2B,2C,2D,2E,2F,30,31,32,33

// DMAR entry

[002]                    PCI Path : 1F,07

// UUID within UEFI table

[016]             UUID Identifier : 00,01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F
```

Each hexadecimal byte should be entered separately, separated by a comma.

4.1.5 Fields Set Automatically

There are several types of ACPI table fields that are set automatically by the compiler. This simplifies the process of ACPI table development by relieving the programmer from these tasks.

Checksums: All ACPI table checksums are computed and inserted automatically. This includes the main checksum that appears in the standard ACPI table header, as well as any additional checksum fields such as the extended checksum that appears in the ACPI 2.0 RSDP.

Table Lengths: All ACPI table lengths are computed and inserted automatically. This includes the master table length that appears in the common ACPI table header, and the length of any internal subtables as applicable.

Examples:

```
[004]                          Table Length : 000000F4

[001]                         Subtable Type : 08 <Platform Interrupt Sources>
[001]                                Length : 10

[001]                         Subtable Type : 01 <Memory Affinity>
[001]                                Length : 28
```

Flags: As described in the previous section, individual flags are aggregated automatically by the compiler and inserted into the ACPI table as the correctly sized and valued integer.

Compiler IDs: The data table compiler automatically inserts the ID and current revision for iASL into the common ACPI table header for each table during compilation.

4.1.6 Special Fields

Reserved Fields: All fields that are declared as *Reserved* by the table definition within the ACPI (or other) specification should be set to zero.

Table Revision: This field in the common ACPI table header is often very important and defines the structure of the remaining table. The developer should take care to ensure that this value is correct and current. This field is *not* set automatically.

The iASL table template generator emits tables with a *TableRevision* that is the latest known value.

Table Signature: There are several table signatures within ACPI that are either different from the table name, or have unusual length:

```
FADT – signature is "FACP".
MADT – signature is "APIC".
RSDP – signature is "RSD PTR " (with trailing space)
```

4.1.7 Generic Fields / Generic Data Types

The following "generic" data types/field names are provided to support tables like the UEFI, which mostly consist of platform-defined data.

UINT8	8-bit unsigned integer (input in hex format)
UINT16	16-bit unsigned integer (input in hex format)
UINT24	24-bit unsigned integer (input in hex format)
UINT32	32-bit unsigned integer (input in hex format)
UINT56	56-bit unsigned integer (input in hex format)
UINT64	64-bit unsigned integer (input in hex format)
String	ASCII quoted string
DevicePath	ASCII patch, similar to String
Unicode	Unicode string (input is an ASCII quoted string)
Buffer	Raw buffer of 8-bit unsigned hex bytes
GUID	ASCII UUID string

Examples:

```
 UINT8 : 11
UINT16 : 1122
UINT24 : 112233
UINT32 : 11223344
UINT56 : 11223344556677
UINT64 : 1122334455667788

    String : "This is a string"
DevicePath : "\PciRoot(0)\Pci(0x1f,1)\Usb(0,0)"
   Unicode : "This string will be encoded to Unicode"

    Buffer : AA 01 32 4C 77
      GUID : 11223344-5566-7788-99aa-bbccddeeff00
```

Example UEFI table with generic data types:

```
/*
 * Intel ACPI Component Architecture
 * iASL Compiler/Disassembler version 20101209-32 [Jan  6 2011]
 * Copyright (c) 2000 - 2011 Intel Corporation
 *
 * Template for [UEFI] ACPI Table
 * Format: [ByteLength]  FieldName : HexFieldValue
 */

[004]                          Signature : "UEFI"    /* UEFI Boot Optimization Table */
[004]                       Table Length : 00000036
[001]                           Revision : 01
[001]                           Checksum : 9B
[006]                             Oem ID : "INTEL "
[008]                       Oem Table ID : "TEMPLATE"
[004]                       Oem Revision : 00000001
[004]                    Asl Compiler ID : "INTL"
[004]              Asl Compiler Revision : 20100528

[016]                    UUID Identifier : 03020100-0504-0706-0809-0A0B0C0D0E0F
[002]                        Data Offset : 0000

                                   UINT8 : ab
                                  UINT16 : cdef
                                  UINT24 : 123456
                                  UINT32 : 01020304
                                  UINT56 : 11223344556677
                                  UINT64 : 0102030405060708

                                  String : "This is a string"
                              DevicePath : "\PCI0\ABCD"
                                 Unicode : "Unicode String"

                                  Buffer : 41 42 43 44 45
                                  String : ""
                                    GUID : 03020100-0504-0706-0809-0A0B0C0D0E0F
```

4.2 Data Table Disassembler

The Data Table Disassembler will disassemble and format any ACPI data table (non-AML table) that is supported. The current set of ACPI Data Tables that are supported by the Data Table disassembler and Data Table compiler are shown below:

```
APIC (MADT)        Multiple APIC Description Table
ASF!               Alert Standard Format table
BOOT               Simple Boot Flag Table
BERT               Boot Error Record Table
CPEP               Corrected Platform Error Polling table
DBGP               Debug Port table
DMAR               DMA Remapping table
ECDT               Embedded Controller Boot Resources Table
EINJ               Error Injection table
ERST               Error Record Serialization Table
FACP (FADT)        Fixed ACPI Description Table
FACS               Firmware ACPI Control Structure
HEST               Hardware Error Source Table
HPET               High Precision Event Timer table
IVRS               I/O Virtualization Reporting Structure
MCFG               PCI Memory Mapped Configuration table
MCHI               Management Controller Host Interface table
MSCT               Maximum System Characteristics Table
RSDP               Root System Description Pointer
RSDT               Root System Description Table
SBST               Smart Battery Specification Table
SLIC               Software Licensing Description Table
SLIT               System Locality Distance Information Table
SPCR               Serial Port Console Redirection table
SPMI               Server Platform Management Interface table
SRAT               System Resource Affinity Table
TCPA               Trusted Computing Platform Alliance table
UEFI               Uefi Boot Optimization Table
WAET               Windows ACPI Emulated devices Table
WDAT               Watchdog Action Table
WDDT               Watchdog Timer Description Table
WDRT               Watchdog Resource Table
XSDT               Extended System Description Table
```

These non-AML ACPI data tables can be “disassembled”, meaning that they are formatted with the individual fields and data. While most ACPI tables found in the field are supported, there may exist a few additional ACPI tables that are not defined in the ACPI specification and are not supported by the disassembler (or compiler.)

4.2.1 Example Output

Example disassembly of an FADT. This example contains a revision 4 FADT, which contains both 32-bit and 64-bit addresses for the ACPI registers.

```
/*
 * Intel ACPI Component Architecture
 * AML Disassembler version 20100528
 *
 * Disassembly of FACP.aml, Thu Jun 17 13:18:03 2010
 *
 * ACPI Data Table [FACP]
 *
 * Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue
 */

[000h 0000   4]                    Signature : "FACP"
[004h 0004   4]                 Table Length : 000000F4
[008h 0008   1]                     Revision : 04
[009h 0009   1]                     Checksum : 9F
[00Ah 0010   6]                       Oem ID : "INTEL"
```

```
[010h 0016  8]                 Oem Table ID : "EXAMPLE"
[018h 0024  4]                 Oem Revision : 00000002
[01Ch 0028  4]              Asl Compiler ID : "INTL"
[020h 0032  4]        Asl Compiler Revision : 20100528

[024h 0036  4]                 FACS Address : 78D22000
[028h 0040  4]                 DSDT Address : 71F61000
[02Ch 0044  1]                        Model : 00
[02Dh 0045  1]                   PM Profile : 04 (Enterprise Server)
[02Eh 0046  2]                SCI Interrupt : 0009
[030h 0048  4]             SMI Command Port : 000000B2
[034h 0052  1]            ACPI Enable Value : A0
[035h 0053  1]           ACPI Disable Value : A1
[036h 0054  1]               S4BIOS Command : 00
[037h 0055  1]              P-State Control : 00
[038h 0056  4]     PM1A Event Block Address : 00000400
[03Ch 0060  4]     PM1B Event Block Address : 00000000
[040h 0064  4]   PM1A Control Block Address : 00000404
[044h 0068  4]   PM1B Control Block Address : 00000000
[048h 0072  4]    PM2 Control Block Address : 00000450
[04Ch 0076  4]       PM Timer Block Address : 00000408
[050h 0080  4]           GPE0 Block Address : 00000420
[054h 0084  4]           GPE1 Block Address : 00000000
[058h 0088  1]       PM1 Event Block Length : 04
[059h 0089  1]     PM1 Control Block Length : 02
[05Ah 0090  1]     PM2 Control Block Length : 01
[05Bh 0091  1]        PM Timer Block Length : 04
[05Ch 0092  1]            GPE0 Block Length : 10
[05Dh 0093  1]            GPE1 Block Length : 00
[05Eh 0094  1]             GPE1 Base Offset : 00
[05Fh 0095  1]                 _CST Support : 00
[060h 0096  2]                   C2 Latency : 0065
[062h 0098  2]                   C3 Latency : 03E9
[064h 0100  2]               CPU Cache Size : 0000
[066h 0102  2]           Cache Flush Stride : 0000
[068h 0104  1]            Duty Cycle Offset : 01
[069h 0105  1]             Duty Cycle Width : 00
[06Ah 0106  1]          RTC Day Alarm Index : 0D
[06Bh 0107  1]        RTC Month Alarm Index : 00
[06Ch 0108  1]            RTC Century Index : 32
[06Dh 0109  2]   Boot Flags (decoded below) : 0001
            Legacy Devices Supported (V2) : 1
         8042 Present on ports 60/64 (V2) : 0
                     VGA Not Present (V4) : 0
                   MSI Not Supported (V4) : 0
             PCIe ASPM Not Supported (V4) : 0
[06Fh 0111  1]                     Reserved : 00
[070h 0112  4]        Flags (decoded below) : 000004A5
     WBINVD instruction is operational (V1) : 1
             WBINVD flushes all caches (V1) : 0
                   All CPUs support C1 (V1) : 1
                 C2 works on MP system (V1) : 0
           Control Method Power Button (V1) : 0
           Control Method Sleep Button (V1) : 1
       RTC wake not in fixed reg space (V1) : 0
           RTC can wake system from S4 (V1) : 1
                       32-bit PM Timer (V1) : 0
                     Docking Supported (V1) : 0
              Reset Register Supported (V2) : 1
                           Sealed Case (V3) : 0
                   Headless - No Video (V3) : 0
       Use native instr after SLP_TYPx (V3) : 0
             PCIEXP_WAK Bits Supported (V4) : 0
                    Use Platform Timer (V4) : 0
              RTC_STS valid on S4 wake (V4) : 0
               Remote Power-on capable (V4) : 0
                Use APIC Cluster Model (V4) : 0
    Use APIC Physical Destination Mode (V4) : 0

[074h 0116 12]               Reset Register : <Generic Address Structure>
[074h 0116  1]                     Space ID : 01 (SystemIO)
[075h 0117  1]                    Bit Width : 08
[076h 0118  1]                   Bit Offset : 00
[077h 0119  1]                 Access Width : 01
[078h 0120  8]                      Address : 0000000000000CF9

[080h 0128  1]         Value to cause reset : 06
[081h 0129  3]                     Reserved : 000000
```

```
[084h 0132   8]               FACS Address : 0000000078D22000
[08Ch 0140   8]               DSDT Address : 0000000071F61000
[094h 0148  12]           PM1A Event Block : <Generic Address Structure>
[094h 0148   1]                   Space ID : 01 (SystemIO)
[095h 0149   1]                  Bit Width : 20
[096h 0150   1]                 Bit Offset : 00
[097h 0151   1]               Access Width : 02
[098h 0152   8]                    Address : 0000000000000400

[0A0h 0160  12]           PM1B Event Block : <Generic Address Structure>
[0A0h 0160   1]                   Space ID : 01 (SystemIO)
[0A1h 0161   1]                  Bit Width : 00
[0A2h 0162   1]                 Bit Offset : 00
[0A3h 0163   1]               Access Width : 00
[0A4h 0164   8]                    Address : 0000000000000000

[0ACh 0172  12]         PM1A Control Block : <Generic Address Structure>
[0ACh 0172   1]                   Space ID : 01 (SystemIO)
[0ADh 0173   1]                  Bit Width : 10
[0AEh 0174   1]                 Bit Offset : 00
[0AFh 0175   1]               Access Width : 02
[0B0h 0176   8]                    Address : 0000000000000404

[0B8h 0184  12]         PM1B Control Block : <Generic Address Structure>
[0B8h 0184   1]                   Space ID : 01 (SystemIO)
[0B9h 0185   1]                  Bit Width : 00
[0BAh 0186   1]                 Bit Offset : 00
[0BBh 0187   1]               Access Width : 00
[0BCh 0188   8]                    Address : 0000000000000000

[0C4h 0196  12]          PM2 Control Block : <Generic Address Structure>
[0C4h 0196   1]                   Space ID : 01 (SystemIO)
[0C5h 0197   1]                  Bit Width : 08
[0C6h 0198   1]                 Bit Offset : 00
[0C7h 0199   1]               Access Width : 00
[0C8h 0200   8]                    Address : 0000000000000450

[0D0h 0208  12]             PM Timer Block : <Generic Address Structure>
[0D0h 0208   1]                   Space ID : 01 (SystemIO)
[0D1h 0209   1]                  Bit Width : 20
[0D2h 0210   1]                 Bit Offset : 00
[0D3h 0211   1]               Access Width : 03
[0D4h 0212   8]                    Address : 0000000000000408

[0DCh 0220  12]                 GPE0 Block : <Generic Address Structure>
[0DCh 0220   1]                   Space ID : 01 (SystemIO)
[0DDh 0221   1]                  Bit Width : 80
[0DEh 0222   1]                 Bit Offset : 00
[0DFh 0223   1]               Access Width : 01
[0E0h 0224   8]                    Address : 0000000000000420

[0E8h 0232  12]                 GPE1 Block : <Generic Address Structure>
[0E8h 0232   1]                   Space ID : 01 (SystemIO)
[0E9h 0233   1]                  Bit Width : 00
[0EAh 0234   1]                 Bit Offset : 00
[0EBh 0235   1]               Access Width : 00
[0ECh 0236   8]                    Address : 0000000000000000
```

4.3 ACPI Table Template Generator

The Table Template Generator is used to create examples for each of the supported ACPI tables. It emits code in a format similar to the ACPI data table disassembler, and can compiled directly via the ACPI data table compiler.

These templates contain examples of each possible subtable as applicable to the particular table. The template can be used as a starting point for actual ACPI table development.

Use "**`iasl –T all`**" to generate a template for every supported table

Example Template file for ECDT:

```
/*
 * Intel ACPI Component Architecture
 * iASL Compiler/Disassembler version 20100528
 *
 * Template for [ECDT] ACPI Table
 * Format: [ByteLength]  FieldName : HexFieldValue
 */

[004]                          Signature : "ECDT"
[004]                       Table Length : 00000042
[001]                           Revision : 01
[001]                           Checksum : 2D
[006]                             Oem ID : "INTEL "
[008]                       Oem Table ID : "TEMPLATE"
[004]                       Oem Revision : 00000001
[004]                    Asl Compiler ID : "INTL"
[004]              Asl Compiler Revision : 20100528

[012]            Command/Status Register : <Generic Address Structure>
[001]                           Space ID : 01 (SystemIO)
[001]                          Bit Width : 08
[001]                         Bit Offset : 00
[001]                       Access Width : 00
[008]                            Address : 0000000000000066

[012]                      Data Register : <Generic Address Structure>
[001]                           Space ID : 01 (SystemIO)
[001]                          Bit Width : 08
[001]                         Bit Offset : 00
[001]                       Access Width : 00
[008]                            Address : 0000000000000062

[004]                                UID : 00000000
[001]                         GPE Number : 09
[001]                           Namepath : ""
```

5 Compiler/Disassembler Operation

The iASL compiler is a command line utility that is invoked to translate one or more ASL source files to corresponding AML binary files or the reverse. The syntax of the various command line options is identical across all platforms.

5.1 Command Line Invocation

The general command line syntax is as follows:

```
iasl [options] file1, file2, … fileN
```

5.2 Wildcard Support

Wildcards are supported on all platforms.

On Windows, wildcard support is implemented within the compiler. For other platforms, it is expected that the shell or command line interpreter will automatically expand wildcards into the **argv** array that is passed to the compiler **main().**

5.3 Command Line Options

All compiler options are specified using the '-' (minus) prefix, regardless of the platform of operation. These options are summarized below, and described in detail after.

```
Global:
  -@<file>        Specify command file
  -I<dir>         Specify additional include directory

General Output:
  -p<prefix>      Specify path/filename prefix for all output files
  -va             Disable all errors and warnings (summary only)
  -vi             Less verbose errors and warnings for use with IDEs
  -vo             Enable optimization comments
  -vr             Disable remarks
  -vs             Disable signon
  -w<1|2|3>       Set warning reporting level

AML Output Files:
  -s<a|c>         Create AML in assembler or C source file (*.asm or *.c)
  -i<a|c>         Create assembler or C include file (*.inc or *.h)
  -t<a|c|s>       Create AML in assembler, C, or ASL hex table (*.hex)

AML Code Generation:
  -oa             Disable all optimizations (compatibility mode)
  -of             Disable constant folding
  -oi             Disable integer optimization to Zero/One/Ones
  -on             Disable named reference string optimization
  -cr             Disable Resource Descriptor error checking
  -r<Revision>    Override table header Revision (1-255)

Listings:
  -l              Create mixed listing file (ASL source and AML) (*.lst)
  -ln             Create namespace file (*.nsp)
  -ls             Create combined source file (expanded includes) (*.src)

ACPI Data Tables:
  -T <Sig>        Create table template file for <Sig> (or "ALL")
  -vt             Create verbose templates (full disassembly)

AML Disassembler:
  -d  [file]      Disassemble or decode binary ACPI table to file (*.dsl)
  -da [f1,f2]     Disassemble multiple tables from single namespace
  -dc [file]      Disassemble AML and immediately compile it
                  (Obtain DSDT from current system if no input file)
  -e  [f1,f2]     Include ACPI table(s) for external symbol resolution
  -2              Emit ACPI 2.0 compatible ASL code
  -g              Get ACPI tables and write to files (*.dat)

Help:
  -h              Additional help and compiler debug options
  -hc             Display operators allowed in constant expressions
  -hr             Display ACPI reserved method names
  -ht             Display currently supported ACPI table names
```

5.3.1 Global Options

These options affect the compiler globally.

-@<file> Read additional command line options from a command file. The format of this text file is one complete option per line.

-I<dir> Specify an additional directory for include files. The directory that contains the source ASL file is searched first. Then, any additional directories specified via this option are searched. This option may be invoked an unlimited number of times. Directories are searched in the order they appear on the command line.

5.3.2 General Output

These options affect the output of errors and warnings.

-p<prefix>	Specify the filename prefix used for all output files, including the .AML file. (This option overrides the output filename specified in the DefinitionBlock of the ASL.)
-va	Disable all errors/warnings/remarks. The compiler signon and compilation summary information are the only messages.
-vi	Provide less verbose errors and warnings in the format required by the MS VC++ environment. This allows the automatic mapping of errors and warnings to the line of ASL source code that caused the message.
-vo	Enable optimization comments in the listing file. A remark/comment is made wherever an optimization has been performed.
-vr	Disable all remark messages.
-vs	Disable the compiler signon.
-w<1|2|3>	Set the warning reporting level.

5.3.3 AML Text Output Files

The compiler always emits a binary AML table. These options allow the compiler to create various text versions of the AML code to simplify the inclusion of the code into a BIOS project.

5.3.3.1 Source Code Files (-s)

These options create files that contain the AML in hex format, with a unique label for each line of the original ASL code. This allows the BIOS to easily dynamically access/modify the ACPI table.

-sa	Create AML in an x86 assembly source code file with the extension .ASM. This option creates a file with a unique label on the AML code for each line of ASL code.
-sc	Create AML in a C source code file with the extension .C. This option creates a file with a unique label on the AML code for each line of ASL code.

5.3.3.2 Source External Declaration Files (-i)

These options create files that contain external declarations for the symbols created by the options in the previous section.

-ia	Create an ASM include file (.INC) that contains external declarations for the symbols produced by the –sa option above.
-ic	Create a C header file (.H) that contains external declarations for the symbols produced by the –sc option above.

5.3.3.3 Hex Source Code Files (-t)

These options create files that contain the AML code in hex format, in a single array.

-ta Create a hex table file with the extension .HEX. This file contains raw AML byte data in hex table format suitable for inclusion into an ASM file.

-tc Create a hex table file with the extension .HEX. This file contains raw AML byte data in hex table format suitable for inclusion into a C file.

-ts Create a hex table file with the extension .HEX. This file contains raw AML byte data in an ASL Buffer object format suitable for inclusion into a ASL file.

5.3.4 AML Bytecode Generation

These options affect the actual AML code that is generated by the compiler.

-oa Disable all optimizations.

-of Disable the constant folding feature.

-oi Disable integer optimizations to the Zero/One/Ones AML opcodes.

-on Disable named reference string optimizations.

-r<Rev> Set the revision number of the table header, overriding the existing revision.

5.3.5 Listings

These options control the listings that are produced by the compiler (as the result of the compilation of an ASL file)

-l Create a listing file with the extension .LST. This file contains intermixed ASL source code and AML byte code so that the AML corresponding to each ASL statement can be examined.

-ln Create a namespace file with a dump of the ACPI namespace and the extension .NSP

-ls Create a combined source file with the extension .SRC. This file combines all include files into a single, large source file.

5.3.6 ACPI Data Tables

-T <Sig> Create an ACPI Data Table template file. Use “ALL” for the signature to create templates for all ACPI tables known by iASL.

-vt Create verbose template file(s). This option creates the template file(s) with the full output of the disassembler, include file offsets and summary raw data.

5.3.7 AML Disassembler

These options are used to invoke and control the behavior of the AML disassembler.

-d [file] Disassemble or decode a binary ACPI to a file (.DSL). Tables that contain AML code are disassembled back to ASL code. Tables that do not contain AML code are decoded and displayed with a description of each field within the table.

-da [f1,f2] Disassemble All. Load all files into a single common namespace, then disassemble each. Similar to –e option, but disassembles all of the input files. Convenient for disassembling all AML files for a given machine (DSDT plus all SSDTs.)

-dc [file] Disassemble a binary AML file and immediately compile it.

-e [f1,f2] Include these extra binary AML tables to assist with external symbol resolution. This option is very useful when attempting to disassemble a table that contains cross-table control method invocations. In these cases, it is difficult or impossible to properly disassemble the method invocation without having the definition of the method present (the important missing data is the number of arguments.)

-2 Emit ACPI 2.0 compatible ASL code.

-g Get ACPI tables from the local machine (availability depends on the host implementation.)

5.3.8 Help

-h Additional help

-hc Display a complete list of all ASL operators that are allowed in constant expressions that can be evaluated at compile time. (This is a list of the Type 3, 4, and 5 operators.)

-hr Display a list of the ACPI predefined names (reserved names.)

-ht Display a list of all supported ACPI tables, both AML and data table.

5.4 Examples

5.4.1 Input ASL

Example input ASL that is used for the output examples below.

```
DefinitionBlock ("", "DSDT", 2, "Intel", "EXAMPLE", 1)
{
    Name (BSTP, Package() {0,1,2,3})

    Method (_BST)
    {
        Store (BSTP, Debug)
        Return (BSTP)
    }
}
```

5.4.2 Output of –tc (make C hex table) Option

This is the output of the –tc option. The entire table is emitted in a single C array.

```
/*
 *
 * Intel ACPI Component Architecture
 * ASL Optimizing Compiler version 20100331 [Mar 31 2010]
 * Copyright (c) 2000 - 2010 Intel Corporation
 * Supports ACPI Specification Revision 4.0
 *
 * Compilation of "dsdt.asl" - Tue Apr 27 14:20:41 2010
 *
 * C source code output
 * AML code block contains 0x45 bytes
 *
 */
unsigned char AmlCode[] =
{
    0x44,0x53,0x44,0x54,0x45,0x00,0x00,0x00,  /* 00000000    "DSDTE..." */
    0x02,0xED,0x49,0x6E,0x74,0x65,0x6C,0x00,  /* 00000008    "..Intel." */
    0x45,0x58,0x41,0x4D,0x50,0x4C,0x45,0x00,  /* 00000010    "EXAMPLE." */
    0x01,0x00,0x00,0x00,0x49,0x4E,0x54,0x4C,  /* 00000018    "....INTL" */
    0x31,0x03,0x10,0x20,0x08,0x42,0x53,0x54,  /* 00000020    "1.. .BST" */
    0x50,0x12,0x08,0x04,0x00,0x01,0x0A,0x02,  /* 00000028    "P......." */
    0x0A,0x03,0x14,0x12,0x5F,0x42,0x53,0x54,  /* 00000030    "...._BST" */
    0x00,0x70,0x42,0x53,0x54,0x50,0x5B,0x31,  /* 00000038    ".pBSTP[1" */
    0xA4,0x42,0x53,0x54,0x50                  /* 00000040    ".BSTP"    */
};
```

5.4.3 Output of –sc (make C source) Option

This is the output of the –sc option. The table is emitted in multiple C arrays, approximatly one array per “block” of ASL code. For example, one array is emitted per control method.

```
/*
 *
 * Intel ACPI Component Architecture
 * ASL Optimizing Compiler version 20090730 [Aug 14 2009]
 * Copyright (C) 2000 - 2009 Intel Corporation
 * Supports ACPI Specification Revision 4.0
 *
 * Compilation of "dsdt.asl" - Fri Aug 14 14:59:46 2009
 *
 */
    /*
     *       1....
     *       2....DefinitionBlock ("", "DSDT", 2, "Intel", "EXAMPLE", 1)
     */
    unsigned char    DSDT_EXAMPLE_Header [] =
    {
        0x44,0x53,0x44,0x54,0x45,0x00,0x00,0x00, /* 00000000    "DSDTE..." */
        0x02,0xF1,0x49,0x6E,0x74,0x65,0x6C,0x00, /* 00000008    "..Intel." */
        0x45,0x58,0x41,0x4D,0x50,0x4C,0x45,0x00, /* 00000010    "EXAMPLE." */
        0x01,0x00,0x00,0x00,0x49,0x4E,0x54,0x4C, /* 00000018    "....INTL" */
        0x30,0x07,0x09,0x20,                     /* 0000001C    "0.. " */
    };

    /*
     *       3....{
     *       4....    Name (BSTP, Package() {0,1,2,3})
     */
    unsigned char    DSDT_EXAMPLE_BSTP [] =
    {
        0x08,0x42,0x53,0x54,0x50,                /* 00000021    ".BSTP" */
        0x12,0x08,0x04,0x00,0x01,0x0A,0x02,0x0A, /* 00000029    "........" */
        0x03,                                    /* 0000002A    "." */
    };

    /*
     *       5....
     *       6....    Method (_BST)
     */
    unsigned char    DSDT_EXAMPLE__BST [] =
    {
        0x14,0x12,0x5F,0x42,0x53,0x54,0x00,      /* 00000031    ".._BST." */

    /*
     *       7....    {
     *       8....        Store (BSTP, Debug)
     */
        0x70,0x42,0x53,0x54,0x50,0x5B,0x31,      /* 00000038    "pBSTP[1" */

    /*
     *       9....        Return (BSTP)
     */
        0xA4,0x42,0x53,0x54,0x50,                /* 0000003D    ".BSTP" */
    /*
     *      10....    }
     *      11....}
     *      12....
     */
    };
```

5.4.4 Output of –ic (make include file) Option

This is the output of the –ic option. It creates external declarations for all of the arrays created by the –sc option above.

```
/*
 *
 * Intel ACPI Component Architecture
 * ASL Optimizing Compiler version 20090730 [Aug 14 2009]
 * Copyright (C) 2000 - 2009 Intel Corporation
 * Supports ACPI Specification Revision 4.0
 *
 * Compilation of "dsdt.asl" - Fri Aug 14 15:05:34 2009
 *
 */
extern unsigned char    DSDT_EXAMPLE_Header [];
extern unsigned char    DSDT_EXAMPLE_BSTP [];
extern unsigned char    DSDT_EXAMPLE__BST [];
```

5.4.5 Output of –l (Listing) Option

This is a standard listing file with intermixed ASL and AML code.

```
Intel ACPI Component Architecture
ASL Optimizing Compiler version 20090730 [Aug 14 2009]
Copyright (C) 2000 - 2009 Intel Corporation
Supports ACPI Specification Revision 4.0

Compilation of "dsdt.asl" - Fri Aug 14 15:08:30 2009

       1....
       2....DefinitionBlock ("", "DSDT", 2, "Intel", "EXAMPLE", 1)

00000000....44 53 44 54 45 00 00 00     "DSDTE..."
00000008....02 F1 49 6E 74 65 6C 00     "..Intel."
00000010....45 58 41 4D 50 4C 45 00     "EXAMPLE."
00000018....01 00 00 00 49 4E 54 4C     "....INTL"
00000020....30 07 09 20 ...........     "0.. "

       3....{
       4....    Name (BSTP, Package() {0,1,2,3})

[****iasl****]
dsdt.asl     4:     Name (BSTP, Package() {0,1,2,3})
Optimize 6033 -                           ^ Integer optimized to single-byte AML
opcode (Zero)

 [****iasl****]
dsdt.asl     4:     Name (BSTP, Package() {0,1,2,3})
Optimize 6033 -                             ^ Integer optimized to single-byte AML
opcode (One)

00000024....08 42 53 54 50 .........    ".BSTP"
00000029....12 08 04 00 01 0A 02 0A     "........"
00000031....03 ....................     "."

       5....
       6....    Method (_BST)

00000032....14 12 5F 42 53 54 00 ...    ".._BST."

       7....    {
       8....        Store (BSTP, Debug)

00000039....70 42 53 54 50 5B 31 ...    "pBSTP[1"

       9....        Return (BSTP)
```

```
00000040....A4 42 53 54 50 .........    ".BSTP"
      10....    }
      11....}
      12....

Summary of errors and warnings

ASL Optimizing Compiler version 20090730 [Aug 14 2009]
ASL Input:  dsdt.asl - 13 lines, 178 bytes, 4 keywords
AML Output: dsdt.aml - 69 bytes, 2 named objects, 2 executable opcodes

Compilation complete. 0 Errors, 0 Warnings, 0 Remarks, 2 Optimizations
```

5.4.6 Output of –ln (Namespace Listing) Option

This is a namespace listing file.

```
Intel ACPI Component Architecture
ASL Optimizing Compiler version 20090730 [Aug 14 2009]
Copyright (C) 2000 - 2009 Intel Corporation
Supports ACPI Specification Revision 4.0

Compilation of "dsdt.asl" - Fri Aug 14 15:08:30 2009

Contents of ACPI Namespace

Count  Depth    Name - Type

    1  [1]      _GPE - Scope
    2  [1]      _PR_ - Scope
    3  [1]      _SB_ - Device
    4  [1]      _SI_ - Scope
    5  [1]      _TZ_ - Thermal
    6  [1]      _REV - Integer
    7  [1]      _OS_ - String
    8  [1]      _GL_ - Mutex
    9  [1]      _OSI - Method
   10  [1]      BSTP - Package       [Initial Length  0x04 elements]
   11  [1]      _BST - Method        [Code Length     0x0011 bytes]

Namespace pathnames

\_GPE
\_PR_
\_SB_
\_SI_
\_TZ_
\_REV
\_OS_
\_GL_
\_OSI
\BSTP
\_BST
```

5.5 Integration Into MS VC++ Environment

This section contains instructions for integrating the iASL compiler into MS VC++ 6.0 development environment.

5.5.1 Integration as a Custom Tool

This procedure adds the iASL compiler as a custom tool that can be used to compile ASL source files. The output is sent to the VC output window.

a) Select Tools->Customize.

b) Select the "Tools" tab.

c) Scroll down to the bottom of the "Menu Contents" window. There you will see an empty rectangle. Click in the rectangle to enter a name for this tool.

d) Type "iASL Compiler" in the box and hit enter. You can now edit the other fields for this new custom tool.

e) Enter the following into the fields:

Command:	`C:\Acpi\iasl.exe`
Arguments:	`-e "$(FilePath)"`
Initial Directory:	`"$(FileDir)"`
Use Output Window:	<Check this option>

"Command" must be the path to wherever you copied the compiler.

"-e" instructs the compiler to produce messages appropriate for VC.

Quotes around FilePath and FileDir enable spaces in filenames.

f) Select "Close".

These steps will add the compiler to the tools menu as a custom tool. By enabling "Use Output Window", you can click on error messages in the output window and the source file and source line will be automatically displayed by VC. Also, you can use F4 to step through the messages and the corresponding source line(s).

5.5.2 Integration into a Project Build

The compiler can be integrated into a project build by using it in the “custom build” step of the project generation. The commands and arguments should be similar to those described above.

6 Generating iASL from Source Code

Generation of the ASL compiler from source code requires these items:

6.1 Required Tools

The *flex* (or *Lex*) lexical analyzer generator

The *Bison (Yacc* replacement) parser generator

An ANSI C compiler

6.2 Required Source Code

There are three major source code components that are required to generate the compiler

The iASL compiler source

The ACPICA Core Subsystem source. In particular, the Namespace Manager component is used to create an internal ACPI namespace and symbol table.), and the AML Interpreter is used to evaluate constant expressions.

The Common source for all ACPI components

ACPICA and iASL source code is available at http://www.acpica.org/downloads/

iASL Windows binary is available at http://www.acpica.org/downloads/binary_tools.php

The source files appear in these directories by default:

Compiler Source:	**Acpica/Source/Compiler**
Common Source:	**Acpica/Source//Common**
Subsystem Source:	**Acpica/Source/Components/**