Application Report
SPRA716 –November 2000

Achieving Zero Overhead With the TMS320 DSP Algorithm Standard IALG Interface

Alan Campbell *European Third Party Organization*

The TMS320™ Algorithm Standard assists digital signal processor (DSP) system designers by removing the barriers to integrating algorithms into all types of systems. A key requirement of the standard is that all algorithms must implement the IALG interface to define their memory requirements, enabling efficient use of on-chip data memories in client applications. However standards are often associated with additional overhead of CPU cycles and memory space. This application note describes the full spectrum of flexibility versus overhead, concluding with an example of a zero overhead framework for a highly static system.

The example chosen for this work was a Standard-Compliant TMS320C5000™ hashing algorithm written by Microlink, a TI third party in Israel specializing in cryptography. Its low memory usage, and MIPS consumption presented a significant challenge since any overhead could noticeably impact the algorithm's performance.

The intended audience is both system integrators and TI third parties. A method of successively reducing overhead is outlined in a series of Code Composer Studio™ (CCS) projects for the system integrator, whilst the third party benefits from the knowledge that their algorithms can be written once and deployed widely.

Contents

1 Introduction 2
2 IALG Memory Interface Functions 3
3 The Algorithm Under Test 6
4 Sequence of Builds 6
5 Build 2 – Worst-case IALG Interface Overhead 7
6 Build 3 – Removing Subsections in the Linker Command File 10
7 Build 4 – Removing the Algorithm Specific API Code 12
8 Build 5 – Creating the Object at Design Time Using a Priori Knowledge 13
9 Build 6 – Avoiding malloc() – Static Construction of the Memory Descriptor Table 15
10 Build 7 – Overlays – Reusing Data Memory 16
11 Build 8 – Overlaying Program and Data – Reclaiming algInit() Memory 18
12 Build 9 – Achieving Zero Overhead 21
13 Summary of Algorithm Standard Performance Improvements 23
14 Conclusion 25
15 References 25

Figures

Figure 1. Overview of Successive Overhead Optimizations Performed 3
Figure 2. Different Levels of Algorithm Interface 5
Figure 3. Memory Map for C5402 Highlighting On-chip DARAM in Data and Program Space 19
Figure 4. Program Memory Optimizations 23
Figure 5. Data Memory Optimizations 24
Figure 6. MIPS Optimizations 24

Tables

Table 1. Application Side Generic APIs 5
Table 2. Performance Figures for the Standard-Compliant MD5 Algorithm 6
Table 3. Microlink's Application Side MD5 Specific APIs 7
Table 4. Additional Overhead for IALG Interface Usage 8
Table 5. Build 2 Project File List 9
Table 6. Additional Overhead for IALG Interface Usage 12
Table 7. Specific Code Savings Made in Build 3 12
Table 8. Additional Overhead for IALG Interface Usage 13
Table 9. Additional Overhead for IALG Interface Usage 14
Table 10. Additional Overhead for IALG Interface Usage 16
Table 11. Additional Overhead Savings 16
Table 12. Additional Overhead for IALG Interface Usage 18
Table 13. Additional Overhead for IALG Interface Usage 20
Table 14. Additional Overhead for IALG Interface Usage 22

1 Introduction

The TI TMS320 Algorithm Standard was officially launched in October 1999 for the C5000 and C6000 platforms. As part of the eXpressDSP software initiative its primary goal is to enable DSP system integrators to better construct applications consisting of algorithm components from potentially different vendors. For TI's third party software vendors, it was also a requirement for a single version of an algorithm to be useful in virtually any application. Many algorithms have now passed the TI DSP Algorithm Compliance procedure and customers are keen to design in the software components. However the question has often been raised "How much overhead does the Standard add?" This application report addresses this issue in relation to the IALG memory interface, which all algorithms must implement.

The IALG interface is a set of functions to define a DSP algorithm's memory resource requirements. It is intended to provide system integrators more freedom in the placement of data buffers – some may be critical and should be placed on-chip, while other less used buffers might be deferred to external memory. This uniform memory management scheme enables multiple algorithms to co-exist without contention in a single application. The aim of this report is to prove that the overhead incurred in implementing these functions can be eliminated in certain types of systems.

The algorithm under test is a C5000 Message Digest 5 (MD5) hashing algorithm often used in the encryption domain. It was written by Microlink, a TI third Party in Israel, and passed compliance checking in July 2000. The reason for choosing this particular algorithm to benchmark was that its

MIPS consumption, program memory and data memory usage, are all very low. Any additional overhead presented by the Standard would clearly be a cause for concern.

The work done in producing this application report focuses on a series of CCS projects, with each new build reducing overhead in a particular area. It follows on from the work of Reference 1 using the same techniques in Builds 2-5 for consistency. There is no build 1 as this was reserved for a non-Standard version, which was not available.

Build 2 demonstrates the worst-case overhead in a flexible, fully dynamic system where algorithms may be created or deleted at any time. Builds 3-5 make several simple optimizations based on the assumption that the intended framework is purely static i.e. memory is allocated once and is used for the remainder of the system’s life.

Builds 6-9 demonstrate new advanced techniques of reclaiming the program and data memory after the initialization phase is complete.

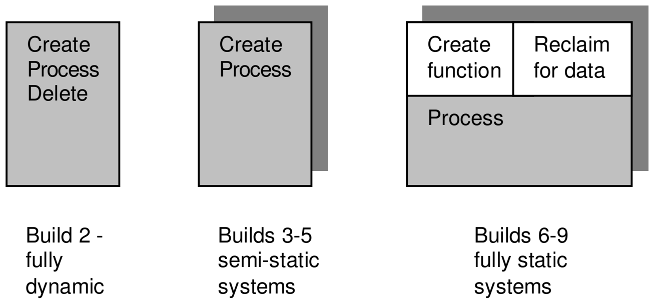

Figure 1. Overview of Successive Overhead Optimizations Performed

2 IALG Memory Interface Functions

The IALG memory interface defines various types and constants with the key element being a global structure of type IALG_Fxns. It contains a set of function pointers, commonly denoted as the v-table. Some of the functions are optional while algAlloc(), algInit(), and algFree() must always be implemented.

```
typedef struct IALG_Fxns {

    Void    *implementationId;

    Void    (*algActivate)(IALG_Handle);

    Int     (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);

    Int     (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);

    Void    (*algDeactivate)(IALG_Handle);

    Int     (*algFree)(IALG_Handle, IALG_MemRec *);

    Int     (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);

    Void    (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);

    Int     (*algNumAlloc)(Void);

} IALG_Fxns;
```

The algAlloc function returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm.

```
Int MODULE_VENDOR_alloc(const IALG_Params *algParams,
                 IALG_Fxns **pf, IALG_MemRec memTab[])
{
    /* Request memory for MODULE object */
    memTab[0].size = sizeof(MODULE_VENDOR_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_EXTERNAL;
    memTab[0].attrs = IALG_PERSIST;

    /*Request memory for additional processing buffers     */
    memTab[1].size = (LEN) * sizeof(Int);
……………………
    return (NUMBUFS);
}
```

Based on the information retrieved from the memTab[ ] descriptor structure, the application allocates the requested memory before calling the algInit() initialization function.

It is the application's responsibility to initialize a pointer, usually of type IALG_Handle, to point to the v-table structure when creating an instance of the algorithm. This provides access to each of the algorithm methods through the function table, without necessarily exposing the vendor's specific function names. The algorithm enables this operation by specifying an instance object containing all of the code's state or context information. Its first field is always of type IALG_Obj which, in turn, makes the IALG functions accessible to the client. The reserved field memTab[0] allows the application to point to the instance object thus implying re-entrancy since all read/write algorithm data memory is encapsulated in a per-channel structure.

algInit() performs all the initialization necessary to complete the run-time creation of an algorithm's instance object. After a successful return from algInit(), the object is ready to be used to process data.

algFree() is the last of the required functions. It is the algorithm's responsibility to make the client aware of the current base addresses and size of each memory block previously requested in algAlloc(), such that the application can delete the instance object and all its buffers, without creating memory leaks. Restoring the memTab[] descriptor structure is also essential housekeeping in the case where the user elects to delete it after initialization.

Clearly it is advisable to have a corresponding set of application side APIs to communicate with TMS320 Standard Algorithms. Two levels of access are defined – generic APIs and specific APIs. The generic APIs are supplied as part of the Developers Kit as an example framework to interface with any compliant algorithm.

Table 1. Application Side Generic APIs

ALG_activate()	Prepare the algorithm to run
ALG_control()	Command and Status mechanism
ALG_create()	Allocate memory and initialize a new algorithm instance
ALG_deactivate()	Prepare the algorithm to be inactive or possibly deleted
ALG_delete()	Remove algorithm instance and deallocate the memory used
ALG_init()	Initialize module other than creating algorithm instance
ALG_exit()	Finalize module other than deleting algorithm instance

The specific API is defined by the algorithm vendor or application writer and is therefore not included in the kit. It typically provides the most convenient access to a particular algorithm as it offers compiler type-safety. The example below shows Microlink's implementation of the application-side MD5_create() function. It is apparent that MD5_create() is simply a wrapper around the ALG_create() function.

```
/*  Create an MD5 instance object (using parameters specified by prms) */
MD5_Handle MD5_create(const IMD5_Fxns *fxns, const MD5_Params *prms)
{
    return ((MD5_Handle)ALG_create((IALG_Fxns *)fxns, NULL, NULL  /*(IALG_Params
*)prms*/));
}
```

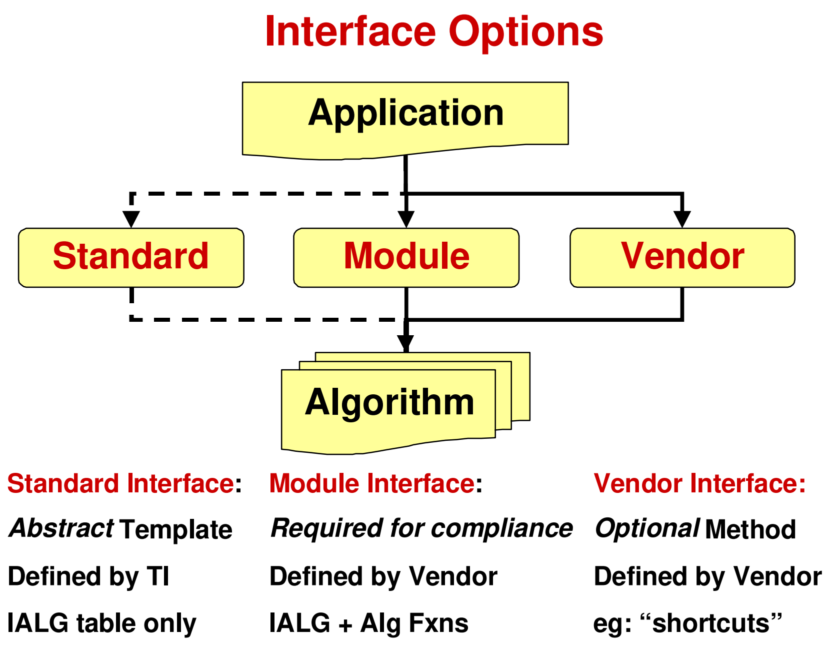

Figure 2. Different Levels of Algorithm Interface

Clearly it is an advantage to use these APIs. Compliance with the Standard's naming conventions makes it immediately clear to the user that he/she is dealing with a Standard-compliant software component.

However the key issue is that all of these application-side APIs are entirely optional. The Standard makes no requirement to use particular framework APIs. It is therefore possible to make several optimizations trading off features like type-safety for savings in program memory. In critical static systems, such decisions are often made.

The final builds also make savings in data memory and MIPS via short-circuiting the v-table. Instead of accessing the main process() function through the v-table, we call the vendor's implementation directly. Again this is a trade-off – the flexibility of potentially swapping in a new vendor's implementation is lost in favor of direct access, in order to gain precious cycles.

3 The Algorithm Under Test

The algorithm under test is a fully compliant C5000 MD5 hashing algorithm written by Microlink, a TI third Party in Israel.

Message Digest 5 is a one-way hash function i.e., it takes an arbitrary length input and generates a unique output. MD5 is used to hash the pass-phrase into the International Data Encryption Algorithm (IDEA) key. MD5 was designed as a successor to MD4. It is slower but more secure.

The performance figures for the algorithm library in isolation are:

Table 2. Performance Figures for the Standard-Compliant MD5 Algorithm

MIPS – Worst Case Cycles/Period	Program Memory (words)	Data Memory (words)
8500	1568	342

It is clear that the MIPS consumption, program memory, and data memory usage are all very low.

The goal was to minimize the overhead (eventually to zero) incurred by conforming to the Algorithm Standard. In addition, no modifications at all were allowed inside the algorithm – all of the optimizations had to be made on the application side. This would demonstrate that the same algorithm could indeed be used in a wide variety of systems.

4 Sequence of Builds

The sequence of optimizations made in successive Code Composer Studio projects will now be described.

The plan of attack was to begin with the most flexible type of system, one in which algorithms may be created or deleted during execution. This represents the worst-case overhead for the Standard since all of the IALG functions implemented by the algorithm are necessary. Memory needs to be allocated and freed at run-time thus consuming MIPS, and program memory for the malloc() (or equivalent) memory allocation calls. Successive optimizations in each Build are then made, as the tradeoff between flexibility and overhead is shifted towards the latter. Build 9 presents the final optimizations towards achieving zero overhead for a highly static (fixed memory and algorithms) system.

To reiterate, Build 1 is omitted for consistency with spra577a.pdf. It shows a non-Standard version for initial benchmarking. A non-Standard MD5 was not available from the vendor, therefore the benchmarking efforts identify the elements particular to the Algorithm Standard and present the reductions in figures at each stage.

All tests were run under Code Composer Studio 1.20 on the Texas Instruments C5402 DSP Starter Kit (DSK) board.

5 Build 2 – Worst-case IALG Interface Overhead

We begin with the worst-case overhead for using the Standard with the MD5 algorithm.

This project uses the algorithm specific API functions both during initialization and execution of the main processing functions. For example, MD5_getDigest() obtains the final message digest after hash processing a block.

```
extern Void  MD5_getDigest  (MD5_Handle handle, XDAS_UInt16 *output);
```

By using algorithm specific types declared in header files, the compiler is able to perform full data-type checking and report any mismatch errors. This provides a level of safety over casting one type to another where bugs may go undetected, as the compiler has no foolproof method for data-type tracking.

Another advantage of this API is the strict naming convention. All of the function prototypes, constants, variables, and data types conform to the conventions enforced by Rule 10 “All modules must follow the naming conventions of the DSP/BIOS for those external declarations disclosed to the client”. The system integrator is immediately aware that he/she is dealing with an Algorithm-Standard component, and the code is easy to read.

Four specific APIs are used in this Build:

Table 3. Microlink’s Application Side MD5 Specific APIs

Algorithm Specific API	Description
MD5_create()	Create an MD5 instance object. Allocate and initialize its data memory.
MD5_delete()	Delete the MD5 instance object and buffers, specified by handle
MD5_process()	Main MD5 processing function
MD5_getDigest()	Obtain the final message digest

In a highly dynamic framework application-side create and delete functions must indeed be present. For example, the system may choose to start a new algorithm on the occurrence of a particular event (e.g. switching from voice to fax channels at the end of the working day). At that time, it may wish to reclaim the data memory used by the algorithm being deactivated. A free/delete function is therefore required.

The elements particular to the Algorithm Standard code were identified from the Project 2 map file as exemplified below:

```
Origin      Size

00001465    00000006     md5_snap_ialg.obj (.text:algDeactivate)
```

A detailed list of the items contributing to the IALG interface overhead is presented in the table below. Successive builds described later will show the improvements made.

Table 4. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x2D4
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
MIPS	N/A	164 cycles

The total size of the program is 9559 sixteen-bit words. This accounts for inclusion of the MD5 algorithm library (1568 words) and the application code to use it. In truth, much of the program image is due to usage of the printf() function from the C5000 Run-Time Support (RTS) library. In order to format its string arguments, printf pulls in a lot of code from other RTS object files and therefore amounts to 7907 words in this example.

The Standard’s 724 words of program memory overhead therefore represents only an 8 percent overhead in this case. However, in a production framework printf debugging information would likely be removed and the overhead percentage would be more significant. It is clear that this metric must be improved.

The MIPS overhead was calculated in simple fashion by comparing the process function MIPS against a direct call to the vendor’s function, MD5_SNAP_process(). An overhead of 164 cycles is incurred due to the additional level of function wrapper MD5_process(), and also as a result of the v-table indirection from the handle via the function table, to the process() function itself.

```
Int  MD5_process(IMD5_Handle handle,XDAS_UInt16 *block,XDAS_Int16 len)
{
    Int  returnValue;


    returnValue = handle->fxns->process(handle,block,len);


    return ((Int )returnValue);
}
```

Data memory overhead is present in the .cinit section due to the static initialization of the v-table. The algorithm fills in this structure with the names of the IALG and extended IALG functions implemented. Nine words are also present in the .const section to provide default parameter values in case the user does not specify any.

The file list in the CCS project is shown below :

Table 5. Build 2 Project File List

Filename	Description
alg_create.c	Default Developer's Kit example APIs for IALG create and delete functions
alg_malloc.c	Default Kit APIs to allocate and delete memory, and activate, deactivate
imd5.c	Application-side file to provide default parameter values
md5_app.c	Application specific APIs to create, delete, and execute processing functions
md5_main.c	The test program
vectors.asm	Boot and Interrupt vectors
md5_snap_ialg.c	Algorithm-side. Implementation of the IALG and processing functions
md5_snap_initexit.c	Algorithm-side. Initialization and Finalization functions. Stub functions.
md5_snap_vtab.c	Algorithm-side. Fills in the v-table with functions which are implemented

It should be noted that the last 3 files were included in the project only to assist with any possible debugging. These form part of the algorithm and are not visible in a typical application. Instead, they are encapsulated in the vendor's library for IP protection.

One point to consider in the dynamic case is the basis for comparison. In a system without the Standard, a format similar to the example below would be required:

ALGO1_create()	ALGO2_create()
ALGO1_process()	ALGO2_process()
ALGO1_delete()	ALGO2_delete()

A separate create function would need to be called for each algorithm since the software may come from potentially different vendors. The same situation exists with deletion.

Now compare this to the Algorithm Standard. We also have algorithm specific create and delete functions but these could easily be rewritten as static inline functions (this is done in a future build)

```
static inline MD5_Handle MD5_create(const IMD5_Fxns *fxns, const MD5_Params *prms)
{
    return ((MD5_Handle)ALG_create((IALG_Fxns *)fxns, NULL, NULL/*(IALG_Params
*)prms*/));
}
```

Any references to MD5_create() are automatically replaced by ALG_create() with the same efficiency as macro substitution. There are now only 2 functions in total for creating and deleting both algorithms, compared to 4 in the non-Standard case.

This is not possible in the original case as there are no vendor-independent standard create, and delete functions to depend upon.

A case could therefore be argued that the Algorithm Standard saves on Program Memory in a highly dynamic system. The gains are greater as more algorithms are added to the system.

6 Build 3 – Removing Subsections in the Linker Command File

This build makes the first assumptions for a static system. As the system integrator we determine our framework characteristics are such that algorithms will be allocated data memory once, and use it forever. No attempts will be made to reclaim it. We can therefore skip the program code for the delete functions.

We further determine that enough internal data memory is available for the algorithm at all times, and that there will never be a need to relocate buffers. These factors allow us to save on the program memory for certain functions.

The NOLOAD output section directive is specified in the linker command file to remove unused code from the program image.

```
.notused {
    *(.text:algActivate)                /* algo side only */
    *(.text:algDeactivate)              /* algo side only */
    *(.text:delete)                     /* app side only */
    *(.text:init)                       /* MD5 module initialization as a whole */
    *(.text:exit)                       /* MD5 module finalization as a whole */
    *(.text:algMoved)                   /* algo side only */
    *(.text:algFree)                    /* algo side only */
    } type = NOLOAD > MYEXT PAGE 0
```

Note that all of these code blocks reside in input subsections e.g. .text:init. These were specified by preprocessor pragma directives, an example of which is shown below:

```
#pragma CODE_SECTION(MD5_create, ".text:create")
```

The primary reason for such pragmas is to ensure that all of the interface functions are fully relocatable, in accordance with Rule 13, “Each of the IALG methods implemented by an algorithm must be independently relocatable”. This allows the system integrator to, for example, defer initialization functions to slower external memory.

However, an additional advantage of pragmas is apparent in this build. They provide function level resolution for the program image. Present linker technology (COFF Linker v3.50) is such that if one function is referenced in a project file, all of the functions in that file are automatically included in the program. The CODE_SECTION pragma combined with the NOLOAD directive circumvents this restriction.

Table 6. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x21E
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
MIPS	N/A	164 cycles

No changes were made in data memory or MIPS in this build. Only the program memory was attacked (changed fields highlighted in gray). 182 words of program memory were recovered from Build 2 to Build 3.

The detailed savings made are listed below.

Table 7. Specific Code Savings Made in Build 3

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory (.notused)	md5_snap_ialg.obj (.text:algActivate)	0x00000006
Program Memory (.notused)	alg_malloc.obj (.text:algActivate)	0x00000018
Program Memory (.notused)	md5_snap_ialg.obj (.text:algDeactivate)	0x00000006
Program Memory (.notused)	alg_malloc.obj (.text:algDeactivate)	0x00000018
Program Memory (.notused)	alg_create.obj (.text:delete)	0x00000018
Program Memory (.notused)	md5_app.obj (.text:delete)	0x00000007
Program Memory (.notused)	md5_snap_initexit.obj (.text:init)	0x00000001
Program Memory (.notused)	alg_malloc.obj (.text:init)	0x00000001
Program Memory (.notused)	md5_app.obj (.text:init)	0x00000001
Program Memory (.notused)	md5_snap_initexit.obj (.text:exit)	0x00000001
Program Memory (.notused)	alg_malloc.obj (.text:exit)	0x00000001
Program Memory (.notused)	md5_app.obj (.text:exit)	0x00000001
Program Memory (.notused)	md5_snap_ialg.obj (.text:algMoved)	0x00000012
Program Memory (.notused)	md5_snap_ialg.obj (.text:algFree)	0x0000001a

7 Build 4 – Removing the Algorithm Specific API Code

In the fourth build, minor optimizations are made to improve both program memory and MIPS. The file md5_app.c provided type-safe wrappers to the generic ALG functions and also called the main processing functions through the v-table.

We now remove this file entirely from the build, and replace the 3 remaining calls with preprocessor macros as shown below:

```
/* define some macros to replace higher level function calls */
#define MD5_CREATE(fxns, prms) \
    (MD5_Handle)ALG_create \
    ((IALG_Fxns *)fxns, NULL, (IALG_Params *)prms)

#define MD5_PROCESS(alg, block, len) \
    ((alg->fxns->process)((MD5_Handle)alg, block, len))

#define MD5_GETDIGEST(handle, digest, output) \
      ( (handle->fxns->getDigest)((IALG_Handle)handle, (&digest)) ); \
      ( memcpy(output, digest, (8*sizeof(short))) )
```

Savings are made in program memory and also in CPU cycles since the overhead of a function call is now removed (improvements compared to previous build are highlighted in gray).

Table 8. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x1EC
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
MIPS	N/A	25 cycles

8 Build 5 – Creating the Object at Design Time Using a Priori Knowledge

This build makes important optimizations based on the assumption that we now have a completely static system. We assume that the algorithm will only ever be instantiated with the same options for data buffer sizes, alignment and type.

This is not an unreasonable assumption. Typical vocoder algorithms use fixed buffer sizes and indeed so too does the Microlink MD5 hashing algorithm. An example in which this might not be the case is a Line Echo Cancellor where the tailspan length parameter dictates the size of the data buffers.

Given that the buffer characteristics are fixed, we now adopt a 2-step approach. The algAlloc() function will always return the same results. It is therefore possible to run this function once a priori, note the returned values, then apply them to the system. Instead of calling algAlloc() in the main application, we simply plug the stored values directly into malloc() memory allocation calls. The constants SIZE_INSTANCEOBJ and SIZE_ALGOCONTEXTBUF represent the values returned by the memTab[].size fields previously returned from an off-line call to algAlloc().

```
    memTab = (IALG_MemRec *)malloc(NUM_ALGOBUFS_REQD * sizeof (IALG_MemRec));

/* I only set what I need to save code space. I know my framework so I don't bother
setting space, attrs fields */

    memTab[0].base = (void *)malloc(SIZE_INSTANCEOBJ);

    memTab[1].base = (void *)malloc(SIZE_ALGOCONTEXTBUF);

    memTab[1].alignment = 2;      /* Algo MD5 needs 32 bit align on context buf */
```

The overhead of the MD5_CREATE macro and consequently the ALG APIs has now been removed. The two files alg_create.c and alg_malloc.c can now be removed from the Project Build. Compared to a total of 9 files in Build 2, there are now only 6. In addition, the .text:algAlloc and .text:algInitObj sections are sent to the NOLOAD section. If we forget to do this, the function code still appears in the program image.

```
.notused {

    *(.text:algActivate)                   /* algo side only */

    *(.text:algDeactivate)                 /* algo side only */

/*    *(.text:delete)   */                     /* app side only */

    *(.text:init)                        /* alg_create, md5_app, md5_snap_initexit share */

    *(.text:exit)                          /* alg_create, the init, exit section for ease */

    *(.text:algMoved)                      /* algo side only */

    *(.text:algFree)                                 /* algo side only */

    *(.text:algAlloc)                                /* algo side only */

    *(.text:algNumAlloc)                             /* algo side only */

    } type = NOLOAD > MYEXT PAGE 0
```

One final improvement is made in this build. Calls to the process and getDigest functions are changed from macros to static inline functions. This has no effect on MIPS – it is simply better programming practice. Macros provide no type-safety and are notoriously error-prone, while statically inlined functions take full advantage of the compiler's type checking and syntax parser. Note that inlining requires the –x2 option to be specified in the Compiler Build options.

Table 9. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x58
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
Data Memory	memTab[] structure	0xa
MIPS	N/A	25 cycles

A side-effect of the huge reduction in program code (highlighted in gray), is that the memTab[] structure is no longer temporary allocation on the stack in ALG_create(). It is now overhead in the data memory of the system, presently on the heap. This problem is resolved in future builds.

9 Build 6 – Avoiding malloc() – Static Construction of the Memory Descriptor Table

The challenging goal of building a zero overhead IALG interface was initially set. New techniques extending beyond Reference 1 must therefore be applied.

This build questions the need for dynamic memory allocation. malloc() (or equivalent) is a complex and expensive operation, a fact which static system integrators are well aware of. The previous build's map file displayed the following entry for the memory module of the C5000 RTS library.

```
Origin          Size
0000261b    0000023e             : memory.obj (.text)
```

At least 574 words of program memory must therefore be reserved for malloc, and possibly more if it depends on other modules. In addition, if we remove the need for malloc, we remove the need for a system data heap. The heap (.sysmem section) is reserved for dynamically allocated data memory and is frequently rounded up to a value greater than our system needs. Whilst no saving in real terms of data memory can be made, in practice we often gain simply by having complete control of our static allocation. For example, the Code Generation Tools default to a 1K word heap, which we no longer require.

Static allocation is now done in a new project file md5_offlinememtab.c. It contains no functions hence does not add to the program image.

```
int memTab0Buf[SIZE_INSTANCEOBJ ];
int memTab1Buf[SIZE_ALGOCONTEXTBUF];


/* good to use keyword const...helps the compiler out! */
const IALG_MemRec offLineMemTab[NUM_ALGOBUFS_REQD] =
{
 {
  SIZEINSTANCEOBJ,          /* memTab[0].size */
    0,                                   /* memTab[0].alignment */
    IALG_EXTERNAL,                /* memTab[0].space */
    IALG_PERSIST,                 /* memTab[0].attrs */
    &memTab0Buf,                  /* memTab[0].base */
 },


 {
```

```
SIZE_ALGOCONTEXTBUF,       /* memTab[1].size */
   2,                              /* memTab[1].alignment */
   IALG_EXTERNAL,                 /* memTab[1].space */
   IALG_PERSIST,                  /* memTab[1].attrs */
   &memTab1Buf                     /* memTab[1].base */
 }
};
```

The data buffers are statically allocated and their addresses are plugged directly into the offLineMemTab[] structure. All other fields were known from an a priori call to algAlloc().

Table 10. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x58
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
Data Memory	memTab[] structure	0xa
MIPS	N/A	25 cycles

It may appear that no savings have been made. However, the gains have been made in the system outside of the IALG interface.

Table 11. Additional Overhead Savings

Removed	Category	Memory Map Section	Size (words) or Number Cycles
X	Program Memory	memory.obj	>= 0x23E
X	Data Memory	.sysmem	0 to 0x400

In addition, 11 other words of .text were saved in the program itself simply by reducing the number of lines required to do the job.

It is fair to say that non-Standard static memory frameworks also avoid the use of malloc() calls, hence the savings may not be fully representative. The aim of this build was primarily to show the final transition to a static system for later builds, and how to set up its memTab[] memory context.

10 Build 7 – Overlays – Reusing Data Memory

Before discussing the optimizations made in this build, let us stop and assess exactly what IALG interface overhead remains.

- Memory Descriptor Table (memTab[]) – despite removing the creation code, and performing static initialization of memTab[], the structure itself is undeniably overhead. With 2 arrays of 5 descriptors, it presents an additional data memory of 10 words.

- Algorithm Instance Object initialization and reset (algInit) – the algorithm's buffer pointers need to point to the memory allocated by the client at memTab[N].base. This functionality must always be present for the system to function as a Standard-based framework. Although not directly related to IALG implementation, additional program memory is also required to initialize the context of other MD5 instance object fields. The .text:algInit section has a size of 88 words.

- Data memory to implement the algorithm's v-table. Ten words in the .cinit section are still occupied by initialization of the IMD5_Fxns global structure.

- Default constants for parameters in imd5.c consume 9 words of .const data memory.

- 25 extra cycles are necessary on each call to the processing functions as a result of v-table indirection.

This build attacks the first item by applying an overlay technique. In a static system, the memory descriptor table is no longer required after the call to algInit(). The instance object is self-sufficient at that time, hence the memTab[] structure data memory can be reused for a different purpose.

In the MD5 static framework it is apparent that the output buffer, an array of 8 words, is not used until after algInit() has completed execution. memTab[] and output[] can therefore be overlayed at the same base address, since they are never used simultaneously.

A C union does the job adequately.

```
typedef union mtabshare {

 IALG_MemRec offLineMemTab[NUM_ALGOBUFS_REQD];

 unsigned short output[8];

} mtabshare;


mtabshare memTabOverlay;
```

In order to view it separately in the map file, it was placed in a separate data section as follows

```
#pragma DATA_SECTION (memTabOverlay, ".secmTabShare")
```

The map file proves that space is allocated only for the larger or the 2 components, in this case, the memTab[] structure. No additional data memory is allocated for the output[8] buffer.

```
.secmTabShare   1    00000b40    0000000a
                   00000b40    0000000a     md5_main.obj (.secmTabShare) [fill = beef]
```

Typically the framework will be larger and contain bigger buffers than in this MD5 example. If we can find large enough application buffers, the entire memTab[] memory can be reclaimed.

Table 12. Additional Overhead for IALG Interface Usage

Category	Memory map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x58
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
Data Memory	memTab[] structure	0x2
MIPS	N/A	25 cycles

11 Build 8 – Overlaying Program and Data – Reclaiming algInit() Memory

The largest contributing factor to the remaining Standard overhead is clearly in the program memory of the algInit() function. It represents the only Standard program code element left to be eliminated, however as stated previously, algInit() must always be called in all frameworks.

It therefore seems impossible to avoid the overhead of this function – yet there is indeed a solution. It relies upon similar techniques applied in Build 7 i.e. given the knowledge that algInit() will only be executed once at start-up of the static system, its program memory can be reused for another purpose. In general, such overlays can be employed in any system possessing an area of RAM – the option is not available with ROM or Flash memory.

The program image is built at link-time and clearly 2 functions cannot be loaded at the same address. The objective is then to try and overlay program and data.

Once again, a block of data must be found which is not used until after the algInit() function has finished execution. The array Dwords[] is used in the application to generate a block for the MD5 algorithm to hash. It is declared as

```
unsigned long Dwords[50];
```

On the C5000 an unsigned long constitutes a 32-bit word, hence the total size of Dwords[] is 100 sixteen-bit words.

The program code size of algInit() is only 88 words. Hence if we can overlay algInit() with Dwords[] then all of its program memory can be reclaimed.

A technique specific to the C5000 architecture is applied to perform the overlay. All C5000 devices (including C54x) contain an area of on-chip DARAM, which is mapped into data space, but may also be mapped into program space.

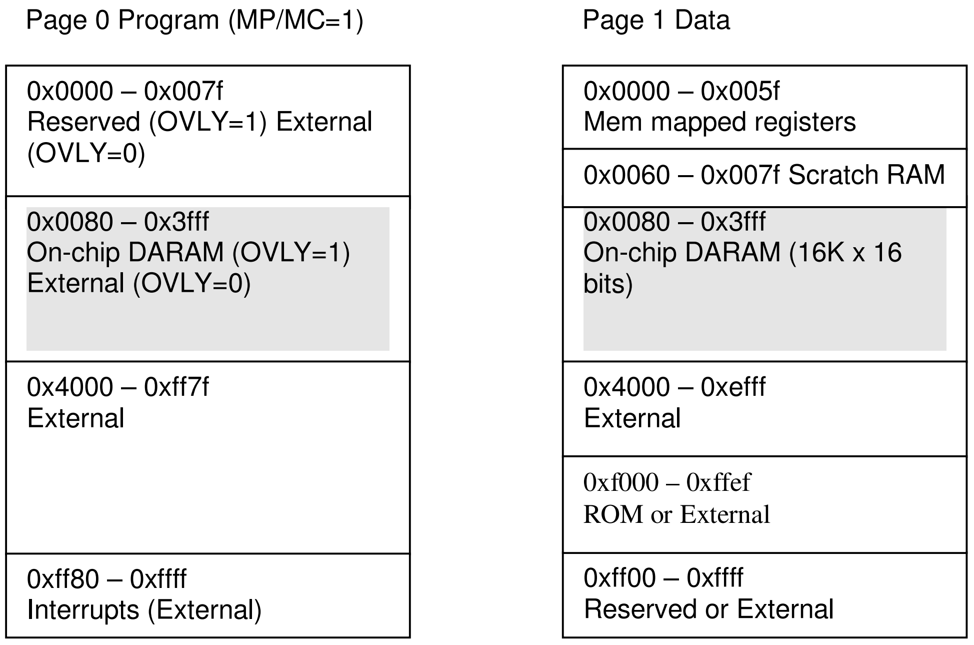

Figure 3. Memory Map for C5402 Highlighting On-chip DARAM in Data and Program Space

A prerequisite for overlays is to set the OVLY bit to 1 in the Processor Mode Status (PMST) Register. OVLY enables on-chip dual-access data RAM blocks to be mapped into program space. In CCS this can be done automatically in a Graphical Extension Language (GEL) file. On CCS startup a value of 0xffe0 is automatically loaded in the PMST register. Bit 5 is the RAM overlay bit hence overlays are enabled by default.

The smallest amount of C5000 on-chip memory is on the C541 which has 4992 words available. It is highly unlikely that an algInit function will ever approach this size and hence, it is feasible to use part of this space for overlays.

The solution was to use the PAGE keyword in the linker command file:

```
MEMORY
{
    PAGE 0: AOBJIMEM:     origin = 0x1200,         len = 0x200
…………………………
    PAGE 1: DWRDIMEM:     origin = 0x1200,        len = 0x200
…………………………
}

SECTIONS
{
 .secDwords:  align = 0x2 fill = 0xbeef {} > DWRDIMEM PAGE 1
 .text:algInit: {} > AOBJIMEM PAGE 0
………………………
}
```

The on-chip DARAM was split in both program (Page 0) and Data (Page 1) spaces. A small area was reserved from address 0x1200 to perform the overlay of algInit program and Dwords[] data memory.

Dwords[] was placed in its own data section to allow independent relocation of the array. It was then linked to output section DWRDIMEM. An alignment of 2 is also specified since it must be located on a 32 bit boundary. Filling the space with a known value (0xbeef) is performed simply to aid debugging.

In program space, the .text:algInit section is mapped to output section AOBJIMEM. This is the same address as DWRDIMEM and uses the same physical memory. Note that the order of declaration is important since .text:algInit is an initialized section. If the Dwords array were linked after the function, it would overwrite and destroy the algInit() program code.

The project map file demonstrates that overlay has been successful. The framework still produced the correct results.

```
.secDwords   1    00001200    00000064
                  00001200    00000064     md5_main.obj (.secDwords) [fill = beef]


.text:algInit   0    00001200    00000058
                  00001200    00000058     md5_snap_ialg.obj (.text:algInit)
```

All of the TMS320 Algorithm Standard program code has now been either removed or reclaimed.

An obvious disadvantage of this technique is the fact that it is ISA dependent, in this case C5000. The PAGE keyword does not exist on C6000 hence slightly different techniques would need to be adopted. In addition, the internal RAM of present C6000 devices is strictly either program or data, not both. Overlays could be applied in external memory however, as this may well be mapped for both code and data e.g. TI's C6211 DSK external memory has such a unified memory map.

Improvements being made in linker technology, particularly within the Visual Linker, may address such concerns in the future. This will also make it significantly easier to implement overlays without detailed knowledge of command file syntax.

The overhead numbers now show a vast improvement.

Table 13. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x0
Data Memory	.cinit	0xd
Data Memory	.bss	0xb
Data Memory	.const	0x9
Data Memory	memTab[] structure	0x2
MIPS	N/A	25 cycles

12 Build 9 – Achieving Zero Overhead

The final build reclaims or removes the remaining data memory overhead, and achieves the theoretical best MIPS.

A preliminary step involved restoring the offLineMemTab[] structure used in Build 6. Build 7 was actually a step sideways in this respect because this statically initialized structure was replaced by an empty one to overlay the output buffer in a C union. We therefore required some extra lines of code in Build 7 to set the fields of data in memTab[], since a union does not accept initialized data.

In addition, a statically initialized structure is easier for the C compiler to optimize, so we should allow for future improvements.

Instead of a union, a simple data overlay scheme is now used. Since memTab[] is no longer useful after completion of algInit(), the output buffer may use the same memory. A one word pointer is sufficient for the output. After memTab[] exhausts its usefulness, and before the output is used, the following assignment is made to point the output to the correct memory:

```
outputPtr = (unsigned short *)&offLineMemTab;
```

Data memory overhead from the Algorithm Standard is still present in the .const section. Removing the imd5.c file from the project can eliminate this. Its function is normally to provide default parameter values when the user fails to supply any. This particular static framework does not use parameters hence 9 words of data memory are recovered.

The final data memory under scrutiny is the entry in .cinit as shown in Build 8’s map file

```
.cinit     0    000034b9    000001c6
                  000034b9    0000000d     md5_snap_vtab.obj (.cinit)
```

The md5_snap_vtab.obj module adds 13 words of overhead. This occurs as a result of v-table initialization. To comply with Rule 12 “All algorithms must implement the IALG interface”, compliant algorithms must fill in a v-table. This typically appears as

```
#define IALGFXNS \
    &MODULE_VENDOR_IALG,                    /* module ID */              \
    MODULE_VENDOR_activate,          /* activate */              \
    MODULE_VENDOR_alloc,                    /* algAlloc */               \
    NULL,                                   /* control (NULL => no control ops) */\
    MODULE_VENDOR_deactivate,        /* deactivate */                    \
    MODULE_VENDOR_free,                     /* free */                          \
    MODULE_VENDOR_initObj,           /* init */                          \
    MODULE_VENDOR_moved,                    /* moved (NULL => not suported) */  \
    MODULE_VENDOR_numAlloc           /* numAlloc (NULL => IALG_DEFMEMRECS) */   \


/*
```

```
 *  ======== MODULE_VENDOR_IMODULE ========
 *  This structure defines VENDOR's implementation of the IMODULE interface
 *  for the MODULE_VENDOR module.
 */
IMODULE_Fxns MODULE_VENDOR_IMODULE = {   /* module_vendor_interface */
    IALGFXNS,
    MODULE_VENDOR_process,
    MODULE_VENDOR_getDigest,
};
```

The v-table allows us to access all of the algorithm's functions through an interface independent of the vendor. If two independent vendors comply with the same interface it is possible to swap out one Standard compliant algorithm for another. It is an excellent feature which undoubtedly gives system integrators more flexibility.

However, it is clear that indirection through the v-table takes extra cycles when calling the critical process functions. If this cannot be tolerated, the framework is free to address the API directly i.e.

```
MD5_SNAP_getDigest((IALG_Handle)handle, &digest);
```

Instead of

```
Handle->fxns->getDigest((IALG_Handle)handle, &digest);
```

The flexibility of swapping one implementation of MD5 out for another is lost, but this is a tradeoff the system designer has made to gain precious cycles.

The v-table has now been entirely bypassed so the file md5_snap_vtab.c can now be removed from the project, bringing the .cinit section overhead to zero. Customers using Standard algorithms archived into a single library will be able to verify this from a map file – no references to functions or data will occur in the v-table implementation hence the corresponding object file will not be included by the linker.

Calling MD5_SNAP_process(), and MD5_SNAP_getDigest() directly finally brings the MIPS overhead incurred by the standard also to zero.

The final overhead table is as follows (changed fields in Build 9 are highlighted in gray):

Table 14. Additional Overhead for IALG Interface Usage

Category	Memory Map Section	Size (words) or Number Cycles
Program Memory	.text + subsections	0x0
Data Memory	.cinit	0x0
Data Memory	.bss	0x1
Data Memory	.const	0x0
Data Memory	memTab[] structure	0x2

MIPS	N/A	0 cycles

It was noted earlier that the 2 word overhead associated with memTab[] was simply a function of this MD5 example framework. Typically, a large enough data buffer can be found to fully overlay memTab[].

The only definite overhead in applying the Algorithm Standard is 1 word in the .bss data memory section. This represents the channel handle pointer, which normally serves two purposes

- v-table traversal for vendor-independent access to both IALG and extended IALG functions
- type-safe, re-entrant access to the channel's instance object.

Since the v-table is no longer used in this framework the first reason is not relevant. The second reason relates to Rule 2 "All algorithms must be re-entrant within a pre-emptive environment". All references to the object are through this pointer to the object which in turn encapsulates all the state information. This encourages re-entrant programming and is therefore enforced. Note that the handle pointer is of type (IMD5_Obj *) i.e. a user defined type, thus providing a high degree of type safety compared to, for example, (Void *).

The advantages the handle brings far outweigh the single word penalty. Indeed it could be argued that even a non-Standard channel-based algorithm might address context data in this fashion as good practice.

Whether or not we deem this to be overhead, it can still be said that the overhead of program memory, data memory, and MIPS is now approximately zero.

13 Summary of Algorithm Standard Performance Improvements

The following diagrams illustrate the progress made in each build towards achieving the end goal of zero overhead for the TMS320 Algorithm Standard.

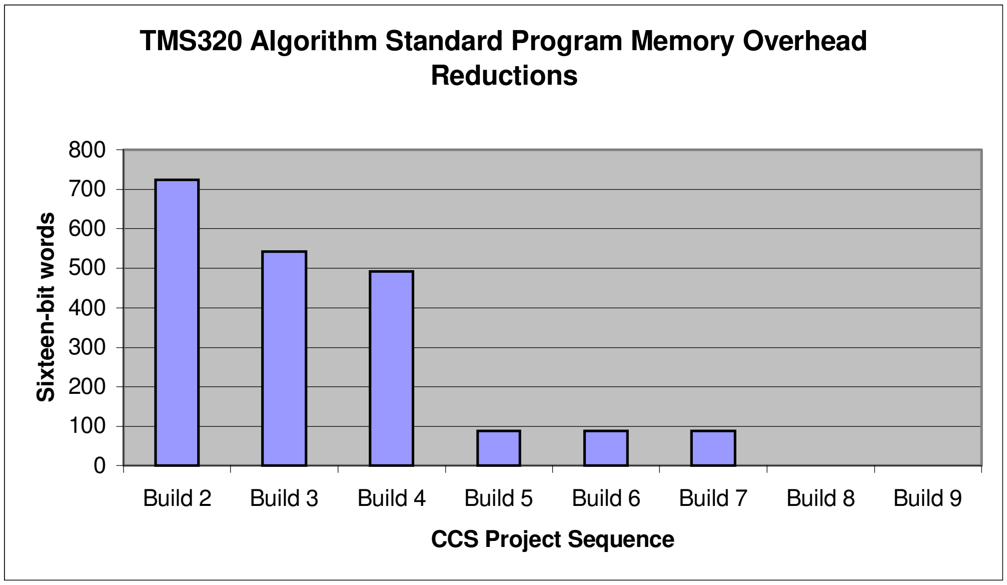

Figure 4. Program Memory Optimizations

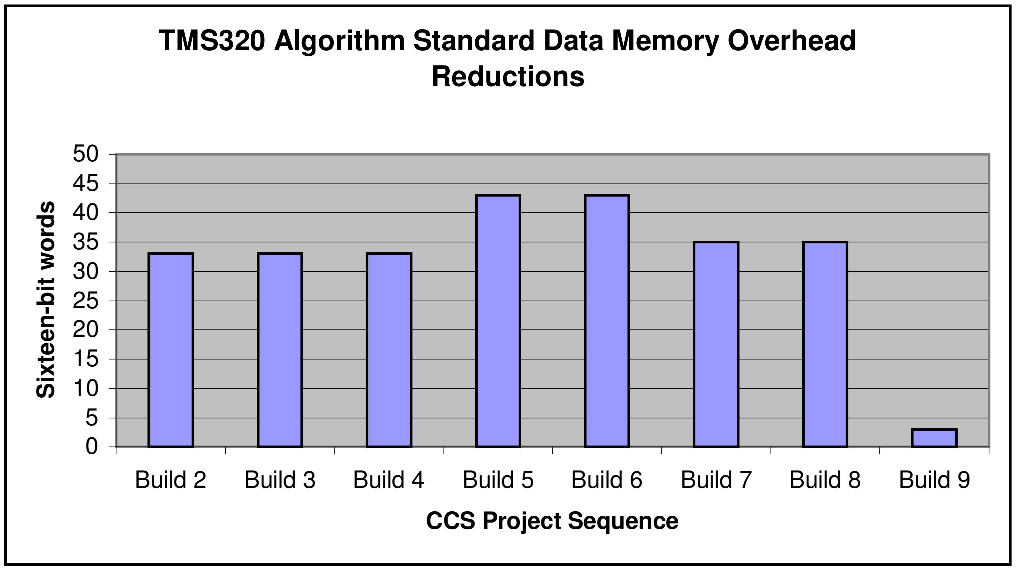

Figure 5. Data Memory Optimizations

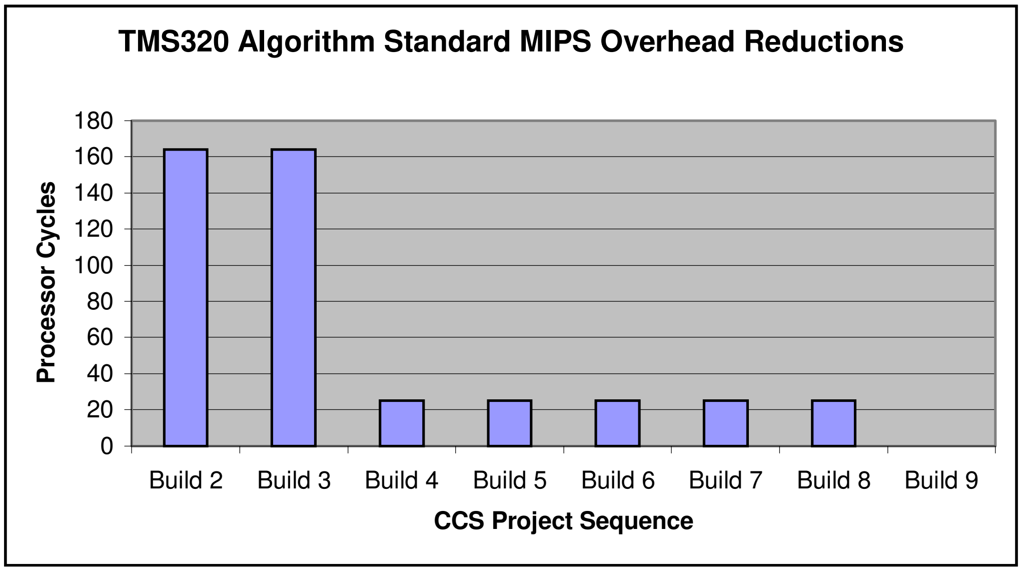

Figure 6. MIPS Optimizations

The most significant program memory gains were made in CCS Projects 3 and 5. The former build relegated unused functions to a NOLOAD section thus removing them from the program image, whilst Build 5 computed algAlloc() off-line, plugged the results into memTab[] directly, and subsequently eliminated all of the supplementary client-side APIs.

The figures for data memory overhead were never particularly large. A typical real-world situation arose in moving from Build 4 to 5; as a result of aggressive program memory optimizations, a minor increase in data memory occurred. Overlay techniques and by-passing the algorithm's v-table rectified the situation bringing the end-result down to just 3 words.

Again the impact of implementing the standard was not significant in the performance of the algorithm. Removing the overhead of function calls, and eventually bypassing the v-table to directly address MD5_SNAP_process(), brought the CPU cycles overhead down from 164 to zero.

14 Conclusion

There are three key conclusions that can be drawn from this application report:

1. Algorithms can indeed be written once and deployed widely. This is a huge advantage for TI's third Party software vendors. Any optimizations can be done entirely on the application side.
2. The same algorithm can be efficiently used in virtually any application. Eight different frameworks (builds 2 – 9) were constructed with each representing a slightly different system. Again, the algorithm remains a constant factor.
3. The Algorithm Standard enables the system integrator to achieve zero overhead. Dramatic optimization of MIPS, program and data memory can be made if the system integrator is building a static system in which the memory configuration is fixed. Alternatively, some overhead may be sacrificed to gain features such as vendor-independence, fully type-safe APIs, and ease of code readability. The feature line is drawn by the system integrator, not the algorithm vendor.

A further improvement to this work would be to simplify the build steps. Linker command file syntax can be tricky, especially when dealing with page overlays, hence it would be advantageous to make use of the graphical Visual Linker Tool.

15 References

1. *Using the eXpressDSP Algorithm Standard in a Static DSP System*, SPRA577
2. *expressDSP Algorithms Standard Rules and Guidelines*, SPRU352
3. *expressDSP Algorithms Standard API Reference*, SPRU360
4. *Making DSP Algorithms Compliant with the eXpressDSP Algorithm Standard*, SPRA579
5. *The eXpressDSP Algorithm Standard - a White Paper*, SPRA581
6. *TMS320C54x Assembly Language Tools*, SPRU102
7. *TMS320C54x DSP Cpu and Peripherals*, SPRU131