Prof. Dr. Norbert Reifschneider

Kernerstraße 7 ~ 74226 Nordheim
Telefon: 0171 / 36 57 275 ~ E-Mail: nreifschneider@t-online.de

IbDR AES Cipher / Decipher Package

for the

ATMEL ATmega MCU Family

Prof. Dr. Norbert Reifschneider
2011 / 2012

<u>Key features:</u>

- Highly optimized for low memory requirements and fast execution time, exclusively written in assembly language and targeted to maximum usage of MCU registers
- Configurable in wide range: For maximum processing speed, for low memory occupancy, for usage of EEPROM memory for constant storage
- Test environment and test patterns taken from AES FIPS PUB 197 included for verification after changes of the code
- Assembler Code available for easy implementation and modification

Content

1 Preface

The IbDR AES Package for the ATMEL ATmega MCU Family is a package of advanced, highly optimized software routines that implement the standard AES cryptographic algorithm for most of the ATMEL ATmega microcontrollers (not all of them, since some minimum memory requirements must be met).

The software is written in assembly language exclusively and can be easily configured via preprocessor constants to achieve goals like low memory requirements or fast execution times.

In particular when using the standard modes of configuration, fast execution times are achieved by excessively using MCU registers (in fact, most time all of the 32 registers are used in parallel) and straight-forward programming strategies. The main subroutines “AES_Cipher” and “AES_InvCipher” for example hold the entire AES State within the MCU registers r0 to r15 without ever storing them out or reloading them from RAM during processing all of the AES Rounds.

The package also includes test patterns from document AES FIPS PUB 197. The test mode can easily be activated by setting a preprocessor constant. This way the software may be tested even if changes were made to it for any reasons.

To include the package into a project, only 3 .asm files and 2 .inc files have to be added to the project. These files will be discussed in detail later in this document. The files are:

Name	**Content**
AES.asm	Assembler source code for all subroutines
AES_Tables.asm	Constant Tables for the AES algorithm: SBox, InvSBox, Rcon- and Galois Values. Usually the constants go to the flash memory, in some configurations the are calculated during initialization of the package or put to the EEPROM
AES_Modules.asm	Experimental and stand-alone-test environment for the software, allows running and testing all subroutines in “AES.asm” without other projects. Not needed in final project
AES_Defines.inc	Assembler include file, contains preprocessor definitions, in particular those to configure the package
AES_TestPast.inc	Contains test patterns taken from AES FIPS PUB 197. If the package is configured for test mode, these patterns are compared automatically with the results of the software. Not needed in final project

2 Some details about AES

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data. It has been adopted by the U.S. government and is now used worldwide. It supersedes the previous encryption standard, DES.

The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

In the USA, AES was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001 after a five-year standardization process in which fifteen competing designs were presented and evaluated before it was selected as the most suitable. It became effective as a Federal government standard on May 26, 2002 after approval by the Secretary of Commerce. It is available in many different encryption packages. AES is the first publicly accessible and open cipher approved by the National Security Agency (NSA) for top secret information.

Originally called Rijndael, the cipher was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and submitted by them to the AES selection process. The name Rijndael (Dutch pronunciation: [ˈrɛindaːl]) is a play on the names of the two inventors.

Strictly speaking, AES is the name of the standard, and the algorithm described is a (restricted) variant of Rijndael. However, in practice the algorithm is also referred to as "AES" (a case of totum pro parte).

AES can work on key lengths of 128, 192 and 256 bit, thus providing flexibility in security level. The present package supports all three key lengths.

The AES algorithm first generates a Key Schedule from the key passed to it which is later used for encrypting and decrypting. The Key Schedule generating algorithm is called “key expansion” and is identical for encrypting and decrypting, resulting in an identical key schedule. The Key Schedule is 176 bytes long for the 128-bit key, 208 bytes long for the 192-bit-key and 240 bytes long for the 256-bit-key.

After generating the Key Schedule encryption or decryption may be executed for an unlimited number of data blocks.

Encryption and decryption both work on 16-byte-blocks of data, called the state. A state box filled with data is passed to the cipher or decipher subroutine along with the Key Schedule generated previously. After processing, the data (encrypted or decrypted) is to be found in the same state box that has been passed to the subroutine. Refer to the subsequent chapters for detailed information about the names and the parameters of each subroutine.

3 Encrypting and Decrypting

Encrypting and decrypting is done by calling the appropriate subroutines of the package. These subroutines are explained in detail in chapter 4, in particular how to pass parameters to them and how to retrieve return values.

If you have chosen one of the modes without computing the Galois field (preprocessor constant "AES_CALC_GALOIS" **not** set), simply call the subroutine "AES_KeyExpand" and subsequently the subroutines "AES_Cipher" for encryption and "AES_InvCipher" for decryption as many times as necessary to complete encryption or decryption of all data. Each call to them will process 16 bytes of data, this is the size of the *State*. Since both subroutines use the same expanded key, they may be called alternately as far as the same key should be applied to encryption and decryption.

In all configuration modes, the "AES_Cipher" and "AES_InvCipher" require a pointer to the RAM memory location where the data to be processed resides. This simply is an array of 16 bytes of data, called the *State*. The address of this array has to be passed to the subroutines in MCU Index Register X. After processing, the encrypted or decrypted data is to be found at the same location.

Via the parameter *Nk*, which is passed to all 3 subroutines in MCU register r16, the subroutines are controlled whether to use the AES-128, AES-192 or the AES-256 variant of the algorithm.

The subroutine "AES_KeyExpand" takes the original key (16, 24 or 32 bytes long for the AES-128, AES-192 or AES-256 algorithm, respectively) as a parameter and performs the so-called Key Expansion. This Key Expansion yields a Key Schedule with a length of 176, 208 or 240 bytes for the AES-128, AES-192 or AES-256 algorithm, respectively. This Key Schedule is used by the "AES_Cipher" and the "AES_InvCipher" in the same way.

Note that the RAM address where the generated Key Schedule is to be stored by the "AES_KeyExpand" subroutine is passed to it as a parameter (in MCU Index Register Y). In contrary, the subroutines "AES_Cipher" and "AES_InvCipher" do not accept such a parameter but use the assembler label "AES_Key_w" that must exist anywhere in the .dseg section of the assembler code and point to the RAM location where the Key Schedule resides. In the standard version of the package, the size of this memory section is set to 240 bytes for the AES-256 version. If the routines are called in AES-128 or AES-192 mode, this memory section remains partially unused. If only the shorter key versions of the AES algorithm should be used, the memory section may be set to a smaller value. For this purpose, change the appropriate line in the "AES.asm" file:

```
;
; AES Key Schedule Field w
; Length of W is 176 for the 128 Bit Key,
; 208 for the 192 Bit Key and 240 for
; the 256 Bit Key. For the shorter Key
; Length, it remains partially unused
;
```

```
AES_Key_w:
.byte 240 / 208 / 176
```

If you have chosen one of the modes with computing the Galois field (preprocessor constant “AES_CALC_GALOIS” **is** set), it is required to call the subroutine “AES_Init” before using the “AES_Cipher” or “AES_InvCipher” subroutines. They both need the Galois Field which is located in flash or EEPROM memory in the Non-Calculate-Galois modes. In Calculate-Galois modes, the Galois Field is computed by a small subroutine and stored into RAM prior to use the “AES_Cipher” or “AES_InvCipher” subroutines when calling the “AES_Init” subroutine. It does not accept any parameters. The subroutines “AES_Cipher” or “AES_InvCipher” in turn are configured automatically when being assembled to read the Galois constants from RAM, EEPROM or flash, whatever is configured by the preprocessor constants.

4 The AES routines

The file “AES.asm” mainly contains the following subroutines (they may call other, small subroutines in turn, which are not documented here in detail):

4.1 AES_ExpandKey

This subroutine generates the AES Key Schedule for the subsequent processing of data. The parameters are:

- MCU Index Register Y (ATmega Register pair r28/r29) must point to a free RAM block of length 176 Bytes for a 128-bit-key, 208 bytes for a 192-bit-key and 240 bytes for a 256-bit-key. A label named “AES_Key_w” must exist anywhere within the RAM area (.dseg area) at this location as the encryption and decryption subroutines take the address for the Key Schedule from there
- MCU Index Register X must point to the given key (16 bytes for the 128-bit-key, 24 bytes for the 192-bit-key and 32 bytes for the 256-bit-key
- MCU Register r16 contains the AES-value *Nk* (refer to document AES FIPS PUB 197 for additional information) which must be 4 for processing a 128-bit-key, 6 for processing a 192-bit-key and 8 for processing a 256-bit-key

The return values after data processing are:

- The generated Key Schedule is to be found in RAM memory at the location where index register Y pointed to on calling the subroutine
- MCU Register r16 contains 0 if no errors occurred, 0xFF otherwise. The only error condition in normal mode is calling the subroutine with a *Nk* value other than 4, 6 or 8. Never-

theless, in test mode an error code is returned if the result calculated from the subroutine does not match the correct Key Schedule specified in the AES FIPS PUB 197 document. This can only happen after changes were made to the assembler source code of the subroutine. Note that, when running in test mode, the key passed to the subroutine is not used, but a standard test key is used instead

4.2 AES_Cipher

This subroutine performs the AES cipher (encryption) algorithm. The parameters are:

- MCU Index Register X (ATmega Register pair r26/r27) must point to the data block (state) where the data to be encrypted resides
- MCU Register r16 contains the AES-value *Nk* (refer to document AES FIPS PUB 197 for additional information) which must be 4 for processing a 128-bit-key, 6 for processing a 192-bit-key and 8 for processing a 256-bit-key

The return values after data processing are:

- The encrypted data is to be found in the same data block (state) where is has been passed to the subroutine (original RAM data has been overwritten)
- MCU Register r16 contains 0 if no errors occurred, 0xFF otherwise. The only error condition in normal mode is calling the subroutine with an *Nk* value other than 4, 6 or 8. Nevertheless, in test mode an error code is returned if the result calculated from the subroutine does not match the correct value specified in the AES FIPS PUB 197 document. This can only happen after changes were made to the assembler source code of the subroutine. Note that, when running in test mode, the data in the state passed to the subroutine is not used, but a standard test pattern is used instead

4.3 AES_InvCipher

This subroutine performs the AES de-cipher (decryption) algorithm. The parameters are:

- MCU Index Register X (ATmega Register pair r26/r27) must point to the data block (state) where the data to be decrypted resides
- MCU Register r16 contains the AES-value *Nk* (refer to document AES FIPS PUB 197 for additional information) which must be 4 for processing a 128-bit-key, 6 for processing a 192-bit-key and 8 for processing a 256-bit-key

The return values after data processing are:

- The decrypted data is to be found in the same data block (state) where is has been passed to the subroutine (original RAM data has been overwritten)
- MCU Register r16 contains 0 if no errors occurred, 0xFF otherwise. The only error condi-

tion in normal mode is calling the subroutine with an *Nk* value other than 4, 6 or 8. Nevertheless, in test mode an error code is returned if the result calculated from the subroutine does not match the correct value specified in the AES FIPS PUB 197 document. This can only happen after changes were made to the assembler source code of the subroutine. Note that, when running in test mode, the data in the state passed to the subroutine is not used, but a standard test pattern is used instead

4.4 AES_Init

This subroutine is only needed when particular configuration settings are used and is only available then (will not be assembled otherwise). It must be called before any other of the encryption/decryption subroutines are used.

An example is using the "AES_CALC_GALOIS" preprocessor setting. In this case, the required Galois field is not stored in flash memory, but the values are computed when executing the "AES_Init" subroutine and moved to the RAM memory. Refer to the subsequent chapters for more information.

The "AES_Init" subroutine does not accept any parameters. Nevertheless, if one of the modes that need it is configured, it uses assembler labels that in turn are only set for these modes. An example is the label "AES_GaloisField" in the .dseg section of the file "AES.asm". It reserves 1.536 Bytes of RAM for the Galois constants that normally reside in the flash memory or alternatively in the EEPROM memory. Only if the configuration "AES_CALC_GALOIS" is set, the label (and the appropriate RAM memory section of course) will be available.

5 Configuration of the package

The package may be individually configured by setting preprocessor constants in the file "AES_Defines.inc". The preprocessor constants are activated simply by setting them using the "#define" preprocessor instruction. The standard "AES_Defines.inc" file contains all constants available but commented out. These constants are:

AES_CALC_GALOIS:

Without this constant, the Galois constants (6 * 256 = 1.536 Bytes) are held in the flash memory. When this preprocessor constant is set, the software does not store the Galois constants in the flash memory but computes them within the initialization routine when executing it, thus freeing 1.536 Byte of memory in the flash memory. Nevertheless, now the same amount of memory in RAM is consumed while executing the package. This RAM block may be used otherwise after termination of the package

AES_TABLES_IN_EEPROM:

When this constant is set, all tables required for the AES algorithm are held in EEPROM memory, thus freeing 2.096 Bytes of memory in the flash memory. This preprocessor constant may be used in conjunction with "AES_CALC_GALOIS". In this case, only 560 Bytes will be occupied in the flash memory while the Galois constants are calculated while executing the initialization routine

AES_DBG_TABLES_IN_EEPROM:

This preprocessor constant is only required when testing the package with the AVR Studio 5 Environment. This version lacks of a method to initialize EEPROM contents for simulation. When this preprocessor constant is set, the tables are automatically stored in the flash memory and copied to the EEPROM when simulating the test software

AES_COMPACT:

This preprocessor constant chooses the compact code version of the package. The resulting code is less than half the size of the standard version (not regarding tables stored in flash which require more space than the executable code), while MCU cycles required for execution are almost twice the ones for the standard version

AES_TEST_MODE:

This preprocessor constant assembles the package in the test mode. In this mode, the subroutines are supplied automatically with test patterns taken from AES FIPS PUB 197 while the calculated results are checked against the given results from the same document. This mode is used preferably with the test environment file "AES_Modules.asm". After return from any of the Cipher / InvCipher subroutines, the value in register r16 will indicate an error condition if the calculated results do not match the expected results from document AES FIPS PUB 197. In this case, the program branches to a certain location where a breakpoint can be set in order to rapdily identify the problem. Note that such problems can only occur if changes were made to the assembler source code

AES_TESTExpandKey_MODE:

This preprocessor constant explicitly checks the routine "AES_KeyExpand". When it is set, the subroutine will check each generated subkey for the final Key Schedule against given true results in document AES FIPS PUB 197. The subroutine "AES_KeyExpand" will return an error code in register r16 if any subkey does not match

5.1 The Configuration Modes in Detail

In this chapter, we will discuss the various configuration modes in detail. Subroutines in round braces are called from the AES subroutines but are not explained in this document. They are listed in the tables to show their memory occupancy only. Other subroutines are listed to show memory occupancy and MCU Cycle requirements for execution.

5.1.1 Standard Configuration Mode

In standard configuration mode, all of the preprocessor constants listed in chapter 5 are removed or commented out. The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Standard, (Straight forward, Speed optimized)**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5			704	560	1.536	4.386	5.289	6.192
(AES_AddRoundKey)				2			82					
AES_InvCipher	32	16	240	5			948	560	1.536	6.173	7.468	8.763
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2			228	560		2.441	2.697	3.229
Total:	**32**	**16**	**240**	**5**	**0**	**0**	**1.976**	**560**	**1.536**			
Sums:	**293**					**0**	**4.072**					

This mode is suitable if sufficient space in the flash memory is available. Its most important benefit is fast execution time (low amount of MCU Cycles required for data processing).

5.1.2 Compact Code

The Compact Code Mode is activated by setting (*#define*-ing) the preprocessor constant “AES_COMPACT”. In this mode the program uses loops whenever possible, computes addresses instead of using constant values in straight-forward-sequences and so forth.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Compact Code (Code Size optimized)**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5			270	560	1.536	9.339	11.236	13.133
(AES_AddRoundKey)				2			32					
AES_InvCipher	32	16	240	5			366	560	1.536	11.459	13.822	16.185
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2			228	560		2.441	2.697	3.229
Total:	**32**	**16**	**240**	**5**	**0**	**0**	**910**	**560**	**1.536**			
Sums:	**293**					**0**	**3.006**					

This mode is suitable when memory limitations exist. Nevertheless, it requires more MCU Cycles for data processing. Refer to modes “AES_CALC_GALOIS” and “AES_TABLES_IN_EEPROM” for freeing even more flash memory.

5.1.3 Standard Mode, Calculate Galois

The “Calculate Galois” mode is activated by setting *(#define*-ing) the preprocessor constant “AES_CALC_GALOIS”.

In this mode, the Galois constants are not held in flash memory but are computed once before AES data processing starts. It frees 1.536 bytes in flash memory, but occupies the same amount in RAM. This mode requires calling the subroutine “AES_Init” before any AES subroutine is called subsequently.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Standard, Calculate Galois**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5	1.536		704	560		4.098	4.937	5.776
(AES_AddRoundKey)				2			82					
AES_InvCipher	32	16	240	5	1.536		948	560		5.597	6.764	7.931
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2			228	560		2.441	2.697	3.229
AES_Init				2	1.536		8	560		162.882	162.882	162.882
AES_CreateGalois				2	1.536		28			162.871	162.871	162.871
AES_GaloisMul				2			28			90	90	90
Total:	**32**	**16**	**240**	**5**	**1.536**	**0**	**2.040**	**558**	**0**			
Sums:			**1.829**			**0**		**2.598**				

This mode is suitable if limitations exist in the flash memory but there is sufficient RAM memory available. Refer to mode “AES_TABLES_IN_EEPROM” for freeing even more flash memory.

5.1.4 Compact Code, Calculate Galois

The “Compact Code, Calculate Galois” mode is activated by setting *(#define*-ing) both the preprocessor constants “AES_COMPACT” and “AES_CALC_GALOIS”.

This mode combines the Compact Code mode explained in 5.1.2 with the Calculate Galois mode explained in 5.1.3.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Compact Code, Calculate Galois**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5	1.536		270	560		9.051	10.884	12.717
(AES_AddRoundKey)				2			32					
AES_InvCipher	32	16	240	5	1.536		366	560		10.883	13.118	15.353
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2			228	560		2.441	2.697	3.229
AES_Init				2			8			162.882	162.882	162.882
AES_CreateGalois				2	1.536		28			162.871	162.871	162.871
AES_GaloisMul				2			28			90	90	90
Total:	**32**	**16**	**240**	**5**	**1.536**	**0**	**974**	**560**	**0**			
Sums:	**1.829**					**0**	**1.534**					

This mode is suitable if flash memory limitations exist but there is sufficient RAM available. Note that execution of the “AES_Init” subroutine is mandatory before calling any AES subroutines and that more MCU Cycles are required for data processing.

5.1.5 Standard Mode, AES Tables in EEPROM

The “AES Tables in EEPROM” mode is activated by setting *(#define*-ing) the preprocessor constant “AES_TABLES_IN_EEPROM”.

In this mode, all AES tables (SBox, InvSBox, Rcon, Galois constants) are stored in EEPROM instead of the flash memory, freeing 2.096 bytes of flash memory. Since this amount of data will be stored now in EEPROM, at least ATmega640 or equivalent is required.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Standard, AES Tables in EEPROM**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5		2.096	736			12.162	14.729	17.296
(AES_AddRoundKey)				2			82					
AES_InvCipher	32	16	240	5		2.096	1.012			18.989	23.068	27.147
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2		2.096	260			4.300	3.913	4.725
AES_ReadEEPR				2		2.096	14					
Total:	**32**	**16**	**240**	**5**	**0**	**2.096**	**2.118**	**0**	**0**			
Sums:	**293**					**2.096**	**2.118**					

This mode is suitable in situations where sufficient EEPROM memory is available while flash memory is low. Note that this mode requires much more MCU cycles since each table data must be read from EEPROM while processing data. If there is not sufficient EEPROM memory available or data processing is too slow, this mode may be combined with “AES_CALC_GALOIS”.

5.1.6 Compact Code, AES Tables in EEPROM

The “Compact Code, AES Tables in EEPROM” mode is activated by setting *(#define*-ing) both the preprocessor constants “AES_COMPACT” and “AES_TABLES_IN_EEPROM”.

This mode is similar to the “Standard Mode, AES Tables in EEPROM” (section 5.1.5) but here the Compact Code version (section 5.1.2) of the assembler code is used.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Compact Code, AES Tables in EEPROM**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5		2.096	286			17.243	20.836	24.429
(AES_AddRoundKey)				2			32					
AES_InvCipher	32	16	240	5		2.096	398			24.547	29.758	34.969
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2		2.096	260			4.300	3.913	4.725
AES_ReadEEPR				2			14					
Total:	**32**	**16**	**240**	**5**	**0**	**2.096**	**1.004**	**0**	**0**			
Sums:	**293**					**2.096**	**1.004**					

This mode is suitable when flash memory is very low and slow execution times are acceptable. It requires even more MCU cycles than the “Standard Mode, AES Tables in EEPROM” mode since now each table data must be read from EEPROM and additionally the Compact Code mode requires more cycles than the standard mode.

5.1.7 Standard Mode, AES Tables in EEPROM, Calculate Galois

The “Standard Mode, AES Tables in EEPROM, Calculate Galois” mode is activated by setting *(#define*-ing) both the preprocessor constants “AES_TABLES_IN_EEPROM” and “AES_CALC_GALOIS”.

This mode is similar to the “Standard Mode, AES Tables in EEPROM” (section 5.1.5) but here the Calculate Galois mode (section 5.1.3) of the assembler code is used in addition.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Standard, AES Tables in EEPROM, Calculate Galois**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5	1.536	560	736			6.978	8.393	9.808
(AES_AddRoundKey)				2			82			61	61	61
AES_InvCipher	32	16	240	5	1.536	560	994			8.477	10.220	11.963
(AES_AddInvRoundKey)				2			14			76	76	76
AES_KeyExpand	32	16	240	2		560	260			3.913	3.913	4.725
AES_ReadEEPR				2			14					
AES_Init				2			24			43.018	43.018	43.018
Total:	**32**	**16**	**240**	**5**	**1.536**	**560**	**2.124**	**0**	**0**			
Sums:	**1.829**					**560**	**2.124**					

In this mode, only the SBox and InvSBox constants are held in EEPROM (560 bytes), while the Galois constants are computed once when executing the “AES_Init” subroutine and stored in the RAM memory. This must be done before calling any of the AES subroutines. Note that the Galois constants (1.536 bytes) now are held in the RAM memory and thus this amount of storage capacity is required there additionally.

Note that this configuration requires far less MCU cycles than the full-EEPROM configuration described in section 5.1.5. The Galois constants need to be accessed more frequently than the “SBox”/”InvSBox” constants and thus moving them to the RAM memory has much more impact on the required MCU cycles than for example moving the “SBox”/”InvSBox” constants there.

5.1.8 Compact Code, AES Tables in EEPROM, Calculate Galois

The “Compact Code, AES Tables in EEPROM, Calculate” mode is activated by setting *(#define*-ing) all the preprocessor constants “AES_COMPACT”, “AES_TABLES_IN_EEPROM” and “AES_CALC_GALOIS”.

This mode is similar to “Standard Mode, AES Tables in EEPROM, Calculate Galois” but here the Compact Code configuration is used additionally.

The following table shows which amount of memory will be used in this configuration and how many MCU cycles are necessary for data processing:

Configuration:	**Compact Code, AES Tables in EEPROM, Calculate Galois**											
Module	**Memory Occupancy**									**MCU Cycles**		
	RAM					**EEPROM**	**Flash**					
	Key	**State**	**Expanded Key**	**Temp Storage**	**Galois**		**Program**	**Tables**	**Galois**	**AES-128**	**AES-192**	**AES-256**
AES_Cipher	32	16	240	5	1.536	560	270			11.771	14.148	16.525
(AES_AddRoundKey)				2			32					
AES_InvCipher	32	16	240	5	1.536	560	366			13.603	16.382	19.161
(AES_AddInvRoundKey)				2			14					
AES_KeyExpand	32	16	240	2		560	260			3.961	3.913	4.725
AES_ReadEEPR				2			14					
AES_CreateGalois				2	1.536		22					
AES_Init				2			24			43.018	43.018	43.018
Total:	**32**	**16**	**240**	**5**	**1.536**	**560**	**1.002**	**0**	**0**			
Sums:	**1.829**					**560**	**1.002**					

This mode requires more MCU cycles for data processing as “Standard Mode, AES Tables in EEPROM, Calculate Galois”, but less capacity in the flash memory for the compact code.

6 Quickstart Instructions

In this chapter, you will learn to set up the package as quickly as possible.

First, decide whether you only want to implement the functional parts of the package or if you want to test it stand alone including testing it or even want to make modifications to it. Of course, you can simply copy all files into your project directory and only add the ones you need for your project.

6.1 Only implementing the package

If you only want to implement the package, copy the files “AES.asm”, “AES_Tables.asm” and “AES_Defines.inc” to your project directory and include them into the project.

Include the file “AES_Defines.inc” into any of your project files in a way that the assembler reads it before the “.asm” files are read.

In the first lines of the “AES.asm” file, the RAM memory allocations are set (.dseg section). Check if any of the assembler directives there conflict with those of your current project. Normally, this should not happen.

6.2 Running the package stand alone

If you want to run the package stand alone, for example to get an idea how it works, copy additionally the file “AES_Moduls.adm” into your project directory and make it the top module of the project. The project should assemble without errors and should be ready to debug.

6.3 Modifying the package and testing it

If you want to check the results of the AES subroutines or even modify the assembler code the “AES_TestPat.inc” file should be copied into the project directory und included into the (test-) project. This file contains test input stimuli to the AES subroutines and known results both taken from the AES FIPS PUB 197 document.

You can now set the preprocessor constants “AES_TEST_MODE” or “AES_TEST-ExpandKey_MODE”.

6.3.1 AES_TEST_MODE

If the preprocessor constant “AES_TEST_MODE” is set, all regular input to the subroutines is ignored. As usual, you have to call the subroutines “AES_Cipher” or “AES_InvCipher”. Instead of processing the State data passed to it, a test pattern is loaded from the “AES_TestPat.inc” file within the subroutines into the State. There are different test patterns for AES-128, AES-192 and AES-256. When finished, the result is compared with a known result which is also loaded from the “AES_TestPat.inc” file. If the computed result does not match the correct value, the subroutines return the error code 0xFF in MCU register r16, otherwise 0 to denote the error-free case.

Note: There is only one situation where the subroutines “AES_KeyExpand”, “AES_Cipher” and “AES_InvCipher” return an error code. This is only true if a *Nk* value other than 4, 6 or 8 is passed to them in MCU register r16 to control the modes AES-128, AES-192 or AES-256, respectively.

6.3.2 AES_TESTExpandKey_MODE

This test mode has been designed to check the correct computing of the Key Schedule by the subroutine “AES_KeyExpand”. As with the “AES_TEST_MODE” test mode, the original key passed to the subroutine “AES_KeyExpand” will not be used. Instead, test keys will be read from the file “AES_TestPat.inc” and processed. Again, there are different test keys for AES-128, AES-192 and AES-256. When processing is done, the generated Key Schedule is compared with the known result taken from the file “AES_TestPat.inc”.