ITU-T V.42bis Data Dictionary Search on the StarCore™ SC140/SC1400 Cores

By Emilian Medve[1]

This application note presents a StarCore™ SC140/SC1400 core speed-optimized implementation of the data dictionary search algorithm used in the ITU-T V.42bis Recommendation [1]. The V.42bis algorithm is a data compression procedure for use with ITU-T V-series data communications equipment (DCEs) (modems) using error correcting procedures. ITU V.42 and ITU V.42bis are the primary methods for error control and data compression for modem-to-modem communication.

DCEs using Microcom networking protocol MNP5 are several times faster than data transmissions using no data compression (V.42bis) or error correction (V.42), depending on the transmitted data type. Files that are already compressed seem to contain less redundant data and may therefore take longer to transmit using data compression than they would if no data compression were used.

This application note is addressed to StarCore SC140/SC1400 developers. The development tools used in this application include the Metrowerks® StarCore Enterprise C compiler and the Metrowerks CodeWarrior® for StarCore.

CONTENTS

1. This application note was co-originated with Cliona O’Connell, LAKE Datacomms, Ltd., and is based on the LAKE Datacomms Ltd. GENUS family V.42bis software module.

1 ITU-T V.42bis Overview

The ITU-T V.42 (error correction) and ITU-T V.42bis (compression) Recommendations are data link layer protocols to transmit asynchronous data on the general switched telephone network (GSTN). Once a V.42 connection is established, the modems attempt a data compression scheme using V.42bis. For correct operation of the data compression function, an error correcting procedure must be implemented between the two entities using the ITU-T V.42 Recommendation. For V-series recommendations, the Link Access Procedure for Modems (LAPM) error correcting procedures defined in the ITU-T V.42 Recommendation [2] or the error correcting procedures in ITU-T V.120 Recommendation [3] must be implemented. The most important features of the ITU-T V.42bis Recommendation are as follows:

- *Automatic mode switching*. V.42bis adapts to the changing characteristics of a data stream. V.42bis starts in a transparent mode (no compression) and switches to compression mode only when it is effective to do so. Switching back to transparent mode is also automatic when compression is no longer effective. In transparent mode, ASCII characters are sent. In compression mode, codes of 9 bits or more are sent to represent a string of ASCII characters.
- *Reuse of code dictionary*. Further adaptation to changing data characteristics is achieved through reuse of the code dictionary. When the code dictionary is filled, old codes with a low use rate are automatically replaced with new codes.
- *Low transmission overhead*. Encode and decode peer entities build their own code dictionaries, and the code dictionary is not exchanged between peer users. Only a few overhead control codes exist to notify a peer of changes, such as a move from transparent to compression mode.
- *Interworking*. When both end points have V.42bis, operating parameters such as direction of compression, maximum string length, and maximum dictionary size can be negotiated. Provisions are also made in V.42bis to work with equipment that does not have V.42bis capability.

2 Data Dictionary

V.42bis uses a variant of the Lempel-Ziv-Welch (LZW) algorithm. The LZW scheme is a dictionary-based algorithm that attains an average of 6:1 compression, depending on the type of data. Text files are easier to compress than binary files or graphics files. The algorithm encodes a string of characters read from the Data Terminal Equipment (DTE) as a fixed-length codeword. The strings are stored in dictionaries that are dynamically updated during normal computer operation. The data compression function contains two dictionaries, one maintained by the data compression encoder for use in compressing data received from the DTE and one maintained by the data compression decoder for use in the decoding data received from the error control function. The dictionary functions are as follows:

- *String matching*. A sequence of characters is read from the DTE, and the dictionary is searched for the resulting string;
- *Updating*. A new string is added to the dictionary;
- *Deletion*. Infrequently used strings are removed in order to reuse storage capacity.

The dictionary stores strings for use in the encoding, and the decoding process may be logically represented as an abstract data structure. The dictionary can be considered to contain a set of trees, each with a root corresponding to a character in the alphabet. With 8-bit characters, there are 256 trees.

A tree represents the set of known strings beginning with one specific character, and each node in the tree represents one string in the set. A node with no dependent nodes, represented by the hierarchically lower level in the tree, is a leaf node. A leaf node represents the last character in a string. A node with no parent, represented by the hierarchically higher level in the tree, is a root node. A root node represents the first character in a string.

Each node has an associated codeword that uniquely identifies it. The assignment of codewords within the encoder dictionary of a data compression function is equivalent to the corresponding assignment of codewords within the decoder dictionary of the peer data compression function in the remote DCE. The codeword thus provides a reversible encoding of a string.

1. The data dictionary search algorithm matches a sequence of characters (string) with a dictionary entry. The procedure starts with a single character representing the first character in the string. The following steps are then applied:
2. A string is formed from the first character.
3. If the string matches a dictionary entry and the entry is not the entry created by the last invocation of the string matching procedure, the next character is read and appended to the string, and this step is repeated.
4. If the string does not match a dictionary entry or matches the entry created by the last invocation of the string matching procedure, the last character appended to the string is removed. The string thus shortened represents the longest matched string, and the last character represents the unmatched character.

This procedure normally matches the longest string of characters. If the string matching procedure terminates before a longest match is found, the next character from the DTE is treated as the “unmatched character” for the purposes of updating the dictionary and restarting the string matching procedure.

3 Data Dictionary Search Implementation

The encoder and the decoder implement the same data structure for maintaining the dictionary. The dictionary is an array of nodes, and each node has an associated character and a reference to a “child” node. Also, each node has access to an ordered list of “sibling” nodes (reference to the first “sibling” node) that are associated with the same “parent” node. That is, the characters associated with all the siblings are also present in the dictionary as characters that follow the current string.

In this discussion, only the encoder routine for a data dictionary search is analyzed because the decoder function is almost identical. The data dictionary search kernel is a *while* loop that appears identical in the encoder and decoder data dictionary search functions. This data dictionary search kernel executes twice for each transmitted byte of data, once on the encoder side and once on the decoder side. Similar search loop patterns appear in the dictionary update functions, in both the encoder and decoder. Two data dictionary node representations are discussed: an index-based one and a pointer-based one. The index-based representation is the “natural” one, and the pointer-based one is speed optimized. In both cases, the data dictionary is statically allocated to a maximum size.

3.1 Index-Based Representation

In the index-based dictionary representation, the references to the “parent, “child” and “sibling” nodes are represented as indexes. **Example 1** presents the structure of a dictionary node.

Example 1. Index-Based Data Dictionary Representation

```
typedef struct MDCR_V42bis_dictionaryNode_s
{
       unsigned char character;
       unsigned short parent;
       unsigned short child;
       unsigned short sibling;
} MDCR_V42bis_dictionaryNode_t;
```

Example 2 presents the encoder state data structure.

Example 2. Encoder State for Index-Based Data Dictionary Representation

```
typedef struct MDCR_V42bis_E_state_s
{
MDCR_V42bis_dictionaryNode_t dict [MDCR_V42BIS_DICTIONARY_SIZE]; /* 
Encoder data dictionary */
/* ... */
} MDCR_V42bis_E_state_t;
```

Example 3 presents the C implementation of the data dictionary search kernel based on indexes. the `MDCR_V42BIS_NULL` constant has the value –1.

Example 3. C Code for the Index-Based Data Dictionary Search Kernel

```
while(currentCode != MDCR_V42BIS_NULL)
{
       dict = &state->dict[currentCode];
       if(character <= dict->character)
       {
              break;
       }
       lastCode = currentCode;
       currentCode = dict->sibling;
}
```

The assembly code generated by the compiler for the data dictionary search kernel is presented in **Example 4**.

Example 4. Assembly Code for the Index-Based Search Kernel

```
       falign
L3
       tfr      d5,d2                   ;[25]
       asll     #<3,d2                  ;[25]
       move.l   d2,r6                   ;[25]
       nop                              ;[0] AGU stall
       adda     r0,r6                   ;[25]
[
       moveu.b  (r6),d0                 ;[26]
       move.l   r6,(sp-8)               ;[25]
]
```

```
 [
        cmpgt    d0,d1                ;[26]
        adda     #<6,r6               ;[31] B5
]
        bf       <L12                 ;[28]
[
        tfr      d5,d4                ;[30]
        moveu.w  (r6),d5              ;[31]
]
        cmpeq    d5,d6                ;[23]
        bf       <L3                  ;[23]
L12
```

Table 1 presents the performance figures for the index-based data dictionary search kernel.

Table 1. Performance Figures for the C Index-Based Data Dictionary Search Kernel

Node Size (Bytes)	Speed (Cycles)	Size (Bytes)
8	15	30

The code generated by the compiler in **Example 4** shows that computing the address of the next dictionary node requires five cycles. Speed can be improved by reducing the computation time of the address of the next dictionary node. Such a reduction is achieved using pointers in the data dictionary node.

3.2 Pointer-Based Representation

In the pointer-based data dictionary representation, the references to the “parent, “child” and “sibling” nodes are represented as pointers. **Example 5** presents the structure of a dictionary node.

Example 5. Pointer-Based Data Dictionary Representation

```
typedef struct MDCR_V42bis_dictionaryNode_s
{
        unsigned char character;
        struct MDCR_V42bis_dictionaryNode_s * parent;
        struct MDCR_V42bis_dictionaryNode_s * child;
        struct MDCR_V42bis_dictionaryNode_s * sibling;
} MDCR_V42bis_dictionaryNode_t;
```

Example 6 shows the C implementation of the data dictionary search kernel based on pointers.

Example 6. C Code for the Pointer-Based Data Dictionary Search Kernel

```
while((currentCode != NULL) && (character > currentCode->character))
{
        lastCode = currentCode;
        currentCode = currentCode->sibling;
}
```

Example 7 shows the assembly code generated by the compiler for the data dictionary search kernel.

Example 7. ASM Generated Code for the Pointer Based Data Dictionary Search Kernel

```
        falign
L3
        tfra      r4,r1               ;[27]
        adda      #<12,r4             ;[28]
        move.l    (r4),r4             ;[28]
        nop                           ;[0] AGU stall
[
        tsteqa    r4                  ;[25]
        moveu.b   (r4),d5             ;[25] B5
]
        bt        <L11                ;[25]
        cmpgt     d5,d3               ;[25]
        bt        <L3                 ;[25]
L11
```

Table 2 presents the performance figures for the pointer-based data dictionary search kernel.

Table 2. Performance Figures for the C Pointer-Based Data Dictionary Search Kernel

Node Size (Bytes)	Speed (Cycles)	Kernel Size (Bytes)
16	11	20

Example 8 shows the assembly implementation of the data dictionary search kernel based on pointers.

Example 8. ASM Code for the Pointer-Based Data Dictionary Search Kernel

```
        falign
L3
[
        tfra      r4,r1               ;[27]
        adda      #<12,r4             ;[28]
]
        move.l    (r4),r4             ;[28]
        nop                           ;[0] AGU stall
[
        tsteqa    r4                  ;[25]
        moveu.b   (r4),d5             ;[25] B5
]
[
        bt        <L11                ;[25]
        cmpgt     d5,d3               ;[25]
]
        bt        <L3                 ;[25]
L11
```

Table 3 presents the performance figures for the pointer-based data dictionary search kernel.

Table 3. Performance Figures for the ASM Pointer Based Data Dictionary Search Kernel

Node Size (Bytes)	Speed (Cycles)	Kernel Size (Bytes)
16	9	24

4 Results

Table 4 compares the overall performance for the solutions discussed in this document. Notice that the speed-up and memory increase figures are relative to the C index-based data dictionary representation.

Table 4. Overall Data Dictionary Performance Comparison

Version	Node Size (Bytes)	Search Kernel Speed (Cycles)	Search Kernel Size (Bytes)	Node Size Increase (Percent)	Search Kernel Speed-up	Size Increase (Percent)
C Index	8	15	30	—	—	—
C Pointer	16	11	20	100	1.36	–33.33
ASM Pointer	16	9	24	100	1.66	–20

5 Conclusions

The ITU-T V.42bis compression standard includes a data dictionary search algorithm that is difficult to parallelize. The algorithm has a lot of control code and no DSP computation, so it cannot benefit fully from the features of the StarCore SC140 architecture. Therefore, the data structures used are the key to an efficient implementation on the SC140 core, and they must be adapted to the features of the SC140 core. For example, SC140 code that does not use linear addressing of the memory (vector processing) should employ pointers instead of indexes.

The data dictionary node size affects the size of the channel data. In a multi-channel environment, a small value for the channel data size may be a target. Analysis of the data dictionary search kernel reveals that only the sibling member of the data dictionary node is used inside the loop. Therefore, a trade-off can be achieved by representing the sibling as a pointer and the child and parent members as indexes, reducing the data dictionary node size increase to 50 percent while maintaining the same speed for the data dictionary search kernel.

6 References

[1] ITU-T Recommendation V.42bis, January 1990.

[2] ITU-T Recommendation V.42, March 1993.

[3] ITU-T Recommendation V.120, October 1996.

[4] *SC140 DSP Core Reference Manual* (MNSC140CORE).

[5] *SC100 Assembly Language Tools User's Manual (*MNSC100ALT/D).

[6] *SC100 Application Binary Interface Reference Manual (*MNSC100ABI/D).

[7] *CodeWarrior Metrowerks Enterprise C Compiler User's Manual.*

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2270
Rev. 1
11/2004