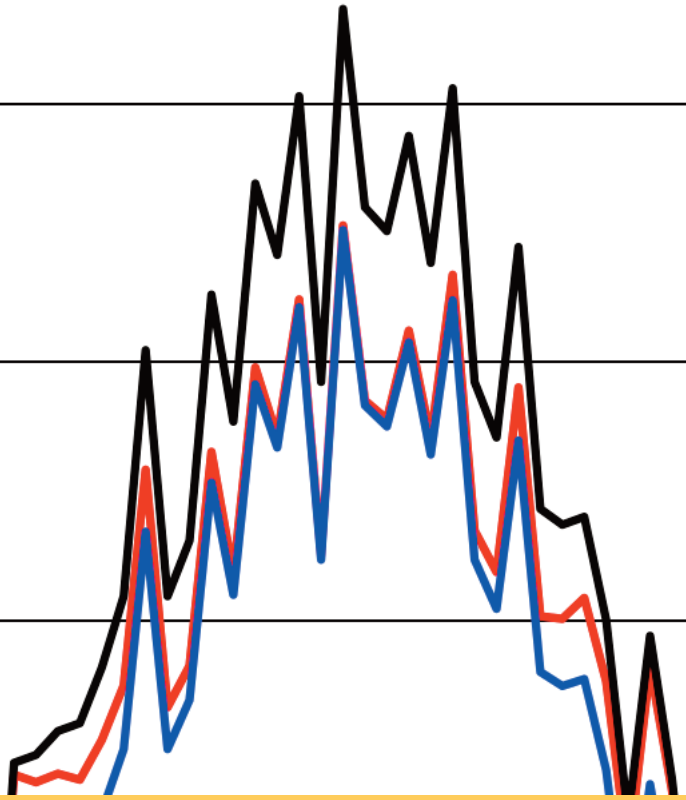
ISSN 2039-7941
anno 2020 numero 422



RAPPORTI TECNICI INGV

Evoluzione del sistema di compressione ACE e applicazione ai dati magnetici

ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

422

REGISTRAZIONE AL TRIBUNALE DI ROMA N.174 | 2014, 23 LUGLIO

ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

ISSN 2039-7941
anno 2020 numero 422

RAPPORTI
TECNICI INGV

Evoluzione del sistema di compressione ACE e applicazione ai dati magnetici

Evolution of ACE compression system and application to magnetic data

Antonino Sicali, Salvatore Consoli, Alfio Amantia, Pasqualino Cappuccio

INGV | Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania - Osservatorio Etneo

Accettato il 21 aprile 2020 | *Accepted 21 April 2020*
Come citare | *How to cite* Sicali A. et al., (2020). Evoluzione del sistema di compressione ACE e applicazione ai dati magnetici. Rapp. Tec. INGV, 422: 1-86.

In copertina Dimensione totale dei dati per l'entropia teorica di Shannon (nero) e rappresentazione dell'informazione (rosso e blu) per il segnale di riferimento attraverso diversi algoritmi | *Cover Total data size for Shannon theoretical entropy (black) and information representation red and blue) for the reference signal by means of different algorithms*

422

INDICE

Riassunto

L'acquisizione di segnali, in particolar modo geofisici, produce una quantità di dati enorme. Ciò dipende ovviamente dal tipo di strumento utilizzato, dal passo di campionamento impiegato e dalla precisione che si vuole raggiungere. Le infrastrutture di comunicazione ricoprono un ruolo importante nell'utilizzo in *real-time* del segnale acquisito, ma non sempre riescono a soddisfare i fabbisogni in termini di velocità di trasmissione e consumo energetico, rendendo impossibile l'utilizzo della telemetria. In tali casi può essere d'aiuto la compressione dei dati che, se realizzata *ad hoc*, riesce a risolvere il problema. L'algoritmo di seguito descritto sfrutta la relazione tra le componenti del segnale a bassa e ad alta frequenza per ridurre la quantità di dati, mettendo in evidenza la quantità d'informazione realmente utile.

Abstract

The signal acquisition, especially in geophysics, produces a huge amount of data. This obviously depends by the instrument used, the sampling rate employed and the precision required. Communication infrastructures play an important role in real-time use of the signal acquired, but they are not always able to meet the needs in terms of transmission speed and energy consumption, so they make impossible using telemetry. In such cases, data compression can help us and, especially if it is made ad hoc, can solve the problem. The algorithm below described uses the relationship between low and high frequency signal components to reduce the amount of data, highlighting the useful information.

Introduzione

Nel 1998, quando fu installata la nuova rete magnetica dell'Etna [Del Negro et al, 1998; 1999; 2002], il sistema di trasmissione basato su tecnologia GSM era agli inizi e una velocità di trasmissione di soli 9600 *baud* rendeva faticoso il *download* dei dati. Inoltre il segnale nei siti remoti non era ottimale e le trasmissioni erano lente e molto disturbate. Occorreva poi ridurre il tempo necessario al *download*, direttamente collegato al consumo energetico, dato che le trasmissioni avvenivano soprattutto di notte, quando non era disponibile l'energia dei pannelli solari, principale fonte energetica utilizzata [Sicali et al., 2020]. Per ridurre il tempo di trasmissione si poteva aumentare la velocità di trasmissione o diminuire il volume dei dati prodotti (cioè la quantità di memoria da essi occupata). Purtroppo la velocità di trasmissione era limitata a 9600 *baud* e non poteva essere incrementata. Bisognava quindi ridurre il volume dei dati, strutturandoli opportunamente. I dati testuali in uscita dai magnetometri possono essere oggetto di diverse ottimizzazioni, dalla semplice codifica binaria alla compressione. Si scelse la compressione rispetto alla semplice codifica binaria poiché con poche righe di codice aggiuntivo si poteva ridurre maggiormente l'occupazione in termini di bit. Ci occuperemo quindi di descrivere i problemi legati alla compressione dei segnali magnetici dal punto di vista strettamente analitico e di dimostrare come sia possibile, aumentando la propria conoscenza sulla sorgente, ottenere il massimo da un algoritmo di compressione scritto appositamente.

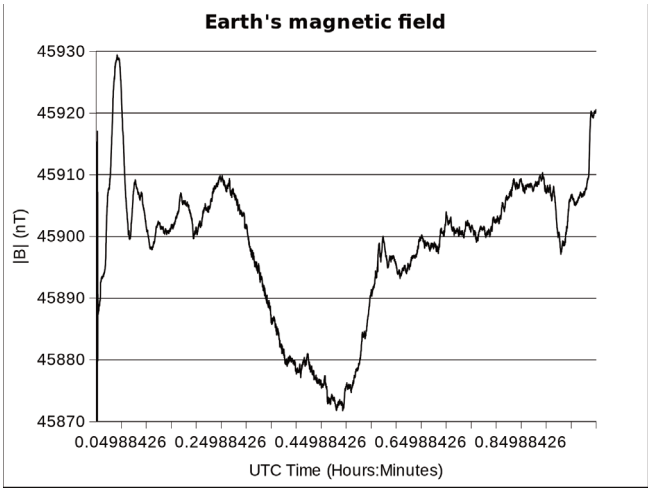
1. Entropia dei segnali magnetici

Nello studio della compressione di dati in formato digitale ricopre un ruolo di primo piano una grandezza chiamata entropia. La conoscenza dell'entropia permette di ottenere algoritmi molto

efficienti e capaci di minimizzare le ridondanze. L'adozione della parola *entropia* si deve a Shannon [Shannon, 1948] che, notando una certa analogia tra la relazione da lui ricavata e la nota grandezza termodinamica, decise di utilizzarne il nome. L'espressione ricavata da *Shannon* includeva un logaritmo e perciò era molto simile alla relazione che definisce l'entropia in termodinamica. *Shannon* dimostrò che l'entropia rappresenta il contenuto informativo per un determinato set di dati ed è direttamente proporzionale alla quantità di informazione ivi contenuta. Dimostrò inoltre che il picco massimo di entropia si ha in presenza di simboli equiprobabili, ovvero di fenomeni come il rumore bianco o variabili casuali uniformemente distribuite. L'entropia va di pari passo ad un'altra grandezza, indicante lo scostamento esistente tra l'occupazione di memoria teorica e reale a parità di informazione: la ridondanza [Shannon, 1948]. La ridondanza rappresenta la quantità di dati che non portano informazione. L'obiettivo degli algoritmi di compressione consiste nell'eliminare la ridondanza (e quindi rappresentare la stessa quantità di informazione utilizzando un minor volume di dati). La difficoltà incontrata dagli algoritmi di compressione è proprio riuscire a separare le due quantità: ridondanza ed entropia. Solitamente l'entropia e la ridondanza sono legate molto bene insieme e ciò ne rende difficile la separazione e la rappresentazione individuale. Come dimostra la storia dell'informazione fin dalle origini nel 1948 si è cercato di ottenere un algoritmo capace di spezzare nettamente il legame tra entropia e ridondanza conservando unicamente l'informazione. Si vedano a proposito algoritmi come lo *Shannon/Fano* [Fano, 1949], *l'Huffman Coding* [Huffman, 1952] ed altri meno conosciuti come la rappresentazione degli interi di *Elias* [Elias, 1975]. L'idea su cui si basano questi algoritmi di codifica è quella di associare un minor numero di bit (parole di codice più brevi) ai simboli più probabili: in questo modo riescono a rappresentare la stessa quantità di informazione utilizzando un minor numero di bit. La conoscenza della sorgente e degli schemi ivi presenti possono ridurre la quantità di ridondanza residua.

Figura 1 Segnale tipico acquisito dalla rete magnetica dell'Etna il giorno 5 Agosto 1998.

Figure 1 Typical signal acquired by Etna's magnetic network on August 5, 1998.



La progettazione di un algoritmo di compressione inizia con l'individuazione della ridondanza presente nella sorgente da digitalizzare [Sicali, 2000]. Nel caso in cui la sorgente fornisca un segnale continuo come quello magnetico si può iniziare calcolandone la derivata e utilizzandola come indice di continuità. Infatti la derivata è inversamente proporzionale alla continuità del segnale e aumenta in maniera spropositata in presenza di segnali discontinui o casuali. La derivata calcolata individua un intervallo di definizione ed è direttamente legata alla quantità d'informazione presente: infatti in presenza di singolarità (*spike*) si creerà un aumento dell'intervallo di definizione della derivata e dell'informazione contenuta. Non è un aumento reale della quantità d'informazione ma una difficoltà di separazione della ridondanza dall'entropia. Difatti esistono molti algoritmi con perdita di dati (*lossy*) che preferiscono preventivamente filtrare le singolarità e l'alta frequenza.

Per calcolare l'entropia si applicherà la relazione di *Shannon* all'intervallo di definizione ricavato, utilizzando la probabilità di ogni singolo valore. Come mostrato dalla figura 2, la distribuzione della derivata ha la forma di una campana centrata sullo zero a rappresentare la tendenza del segnale ad essere continuo e ad evitare variazioni brusche (infinitesimali). Il segnale che possiede tale distribuzione è mostrato nella figura 1 e rappresenta la variazione subita dal campo magnetico terrestre che notoriamente è continuo e privo di brusche variazioni. Nella distribuzione di figura 2 allontanandosi dal centro si entra nelle zone degli *spike* caratterizzate da probabilità molto basse e da un profilo asintotico verso lo zero. Tali zone, nonostante rendano difficoltosa la separazione tra entropia e ridondanza, influiscono poco nel calcolo dell'entropia in quanto prive d'informazione rispetto alle parti più interne. Infatti l'entropia associata alla distribuzione diminuisce all'aumentare dell'altezza della campana e può essere calcolata utilizzando la relazione dell'entropia di *Shannon* [Sicali, 2000]. Ovviamente il valore ricavato rappresenta l'entropia media della sorgente per ciascun simbolo, è specifico solo per il segnale di figura 1 e va ricalcolato ogni volta.

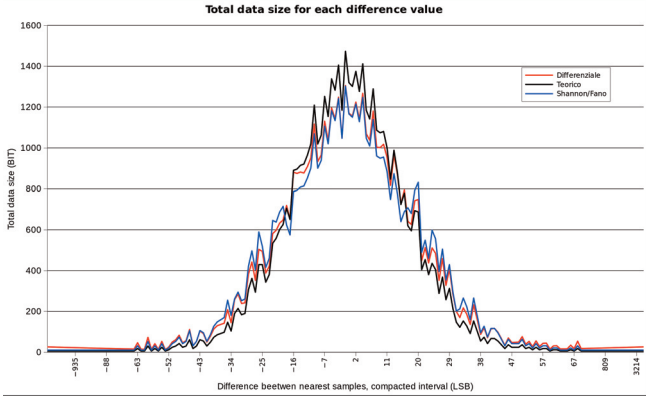


Figura 2 Dimensione totale dei dati per l'entropia teorica di *Shannon* (nero) e rappresentazione dell'informazione (rosso e blu) per il segnale di riferimento attraverso diversi algoritmi. Notare come le curve rossa e blu si trovino sopra quella nera dopo i valori ±18 LSB (*Least Significant Bit)* lungo l'asse x, rilevando una difficoltà di tali algoritmi a seguire le brusche variazioni del segnale.

Figure 2 *Total data size for Shannon theoretical entropy (black) and information representation red and blue) for the reference signal by means of different algorithms. Notice how the red and blue curves are located above the black one after the values ±18 LSB (Least Significant Bit) along the x-axis, detecting a difficulty of these algorithms to follow abrupt variations of the signal.*

2. Algoritmo del 1998 e risultati ottenuti

Il sistema del 1998 era stato studiato per essere eseguito in modo efficiente dai sistemi del tempo, richiedeva poca memoria e poche risorse di calcolo, anche se non raggiungeva perfettamente l'entropia. Le applicazioni erano a 16 bit, non riuscivano a indirizzare molta memoria e avevano risorse di calcolo limitate. Si tendeva a far funzionare le *CPU* a bassa velocità per non consumare troppa energia. Per questo non si riusciva a sfruttare appieno la ridondanza. Ciononostante i risultati sono stati discreti e l'obiettivo di ridurre drasticamente lo spazio di memoria occupato dai dati è stato pienamente raggiunto permettendo alla rete magnetica di funzionare discretamente per oltre un ventennio [Sicali, 2018].
L'algoritmo non eliminava appieno la ridondanza poiché eseguiva una rappresentazione locale suddividendo il numero decimale rappresentante la misura in cifre. Ogni colonna veniva trattata singolarmente. Per ogni colonna venivano elencate le cifre e poi rimappate se queste erano meno di dieci. Questo permetteva di seguire dinamicamente il range ed evidenziare l'entropia, minimizzando l'effetto degli spike e dell'alta frequenza, ovvero le code del grafico della figura 2.
Le prime prove nel 1998 sono state realizzate su 7 sequenze di misure magnetiche lunghe un giorno e campionate a 10 secondi. Ogni singola sequenza è stata compressa più volte fino a ottenere i risultati mostrati nella tabella 1. I risultati confermano la tesi secondo cui una conoscenza maggiore

della sorgente porta a una minor ridondanza residua. Difatti software generici riescono a ottenere risultati peggiori dell'algoritmo specifico. I risultati riportati nella tabella 1 mostrano anche la presenza di ridondanza residua che può essere eliminata dall'evoluzione dell'algoritmo nel 2019.

3. Raggiungimento dell'entropia: algoritmo del 2019

Attualmente non ci sono più problemi di memoria, di calcolo e i sistemi consumano meno energia. L'algoritmo del 1998 è stato completato e nella versione *evolution* del 2019 si preoccupa di raggiungere maggiormente l'entropia riducendo al minimo la ridondanza residua dovuta alla distribuzione della derivata. L'algoritmo considera la misura nella sua interezza, non suddividendola in cifre. Ovviamente considerando la misura nella sua interezza il numero di livelli da trattare sono più delle dieci cifre e quindi è richiesta più memoria e capacità di calcolo. La semplice rimappatura delle cifre viene sostituita dalla costruzione di albero.
Dopo più di vent'anni, i dati prodotti dagli strumenti scientifici continuano a crescere e i problemi di spazio sono sempre attuali nonostante le velocità di trasmissione siano aumentate drasticamente. Per questo si è sentita la necessità di completare il sistema di compressione, anche grazie all'evoluzione tecnologica dei sistemi di calcolo. Le *CPU* sono più efficienti e la memoria non è più limitata a pochi *MB*. L'impiego del sistema nella fase di trasmissione, ma anche durante l'archiviazione, permette di ridurre lo spreco di risorse e il rapporto di compressione nella nuova versione è più che raddoppiato. L'algoritmo utilizzato per la codifica è quello elaborato da *Huffman* [Huffman, 1952] che permette di costruire un albero ottimo rispetto all'algoritmo di *Shannon-Fano* [Fano, 1949]. Mentre *Shannon* dimostra che attraverso una struttura binaria è possibile raggiungere il valore teorico dell'entropia, *Huffman* dimostra che l'albero *bottom-up* è ottimo rispetto a quello *top-down* dell'algoritmo di *Shannon-Fano*. Comunque in entrambi i casi la rappresentazione di ciascun simbolo può risentire di un arrotondamento per eccesso al bit generando ridondanza residua. Applicando il nuovo algoritmo ai dati utilizzati nel 2000 si sono ottenuti i risultati riportati in tabella 2.

File	TXT	ZIP	%	ACE 1998 (TOTAL)	%	ACE 1998 (DELAY)	%	ACE 1998 + ZIP	%
DGL-10	259200	41774	16.12	14257	5.50	9937	3.83	8588	3.31
DGL-11	259200	41877	16.16	15337	5.92	9937	3.83	8622	3.33
DGL-12	259200	41968	16.19	15337	5.92	8857	3.42	6095	2.35
DGL-13	259200	41352	15.95	14257	5.50	7777	3.00	7443	2.87
DGL-14	259200	41705	16.09	16477	6.36	9937	3.83	8732	3.37
DGL-15	259200	43007	16.59	15337	5.92	9937	3.83	8989	3.47

Tabella 1 Risultati ottenuti nel 1998 con diversi tipi di sistemi di compressione col relativo residuo percentuale.
Table 1 *Results obtained in 1998 by means of different types of compression systems with the relative percentage residual.*

I risultati sono concordi nel dimostrare che esiste una parte di ridondanza residua misurabile in una quantità variabile tra il 17% e il 30%. Dal 1998 la rete magnetica è stata oggetto di molte ristrutturazioni e variazioni e da allora non sono state più effettuate prove analitiche, per questo

si è deciso di rifare le prove su altri segnali più recenti. La tabella 3 riporta quindi i nuovi test eseguiti su alcune sequenze di dati nuovi del 2019. La lunghezza media del file è aumentata nonostante la finestra di misura sia sempre 1 giorno. L'intervallo di campionamento è passato da 10 a 5 secondi aumentando il rumore di fondo, che ha peggiorato leggermente le prestazioni.

File	TXT	ZIP	%	ACE 1998	%	BZIP2	%	ACE 2019	%
DGL-10	259200	41774	16.12	9937	3.83	27340	10.55	7606	2.93
DGL-11	259200	41877	16.16	9937	3.83	27410	10.57	7246	2.79
DGL-12	259200	41968	16.19	8857	3.42	27899	10.76	7352	2.84
DGL-13	259200	41352	15.95	7777	3.00	27156	10.48	6198	2.39
DGL-14	259200	41705	16.09	9937	3.83	27661	10.67	7438	2.87
DGL-15	259200	43007	16.59	9937	3.83	28459	10.98	6976	2.69
DGL-16	259200	42820	16.52	8857	3.42	28195	10.88	6942	2.68

Tabella 2 Risultati ottenuti con diversi tipi di sistemi di compressione sui dati del 1998 col relativo residuo percentuale.
Table 2 *Results obtained by means of different types of compression systems on 1998 data with the relative residual percentage.*

File	TXT	ZIP	%	ACE 1998	%	BZIP2	%	ACE 2019	%
DGL-1	725282	105727	14.57	49763	6.86	71320	9.83	34054	4.69
DGL-2	594688	89166	14.99	50587	8.50	59717	10.04	30934	5.20
CSR-1	794880	123483	15.53	54492	6.85	83702	10.53	37471	4.71
CSR-2	794880	124935	15.71	54492	6.85	84216	10.59	37385	4.70
CST-1	930744	135924	14.60	56547	6.07	86779	9.32	37478	4.02
CST-2	931716	133617	14.34	56607	6.07	83826	8.99	40362	4.33

Tabella 3 Risultati ottenuti con diversi tipi di sistemi di compressione su nuovi dati del 2019, col relativo residuo percentuale.
Table 3 *Results obtained by means of different types of compression systems on new 2019, data with the relative percentage residual.*

4. Descrizione e utilizzo del software

Il software originale è scritto in parte in *ASSEMBLY* per ottimizzare ulteriormente le risorse utilizzate e può essere eseguito solo su processori con set d'istruzioni *x86*. Attualmente il software è scritto interamente in *C/C++* e può essere eseguito in qualsiasi ambiente abbia un compilatore di *C/C++* e anche su processori diversi. Nell'appendice A5 è riportato un esempio di codice applicativo. L'interfaccia della classe contiene due funzioni: *compress* e *decompress* che si occupano rispettivamente della compressione e decompressione del segnale. Le due funzioni lavorano direttamente sui file di testo e ricevono come parametro principale un array di

definizione necessario per l'interpretazione del segnale. Tale array può essere fornito alla funzione *compress* dall'utente oppure eventualmente verrà ricavato direttamente eseguendo un riconoscimento delle misure presenti nel file. Alla funzione *decompress* l'array viene fornito dal file in ingresso. L'array di definizione delle misure è composto da una serie di costanti che descrivono in tutte le sue parti la stringa d'ingresso. Le costanti si occupano prevalentemente dell'interpretazione dei dati nella fase della compressione e della ricostruzione della stringa di misura durante la decompressione. Per preservare i dati da eventuali errori viene eseguito un test, calcolando un *checksum*, per una eventuale archiviazione o trasmissione non compressa a causa di problemi software (*bugs*). L'interfaccia della classe prevede anche una funzione *parser* che permette di creare la stringa di definizione attraverso un esempio di misura. Tale funzione è molto utile quando si ha una varietà non indifferente di strumentazioni in cui le misure differiscono anche per poco tra loro. Oltre alle due funzioni che lavorano direttamente sui file, la classe può lavorare in modalità asincrona in collaborazione con il resto del software soprattutto se il *datalogger* di acquisizione non prevede un sistema operativo multi programmato [Sicali, 2018].
Per abilitare o disabilitare l'algoritmo del 2019 si è utilizzata la definizione *_ACE_EVOLUTION*. In presenza di tale definizione il codice viene compilato per eseguire il nuovo algoritmo. L'interfaccia della classe non ha subito variazioni, solo la memoria utilizzata è aumentata sensibilmente e pochi sistemi potrebbero essere capaci di eseguirlo, a differenza dell'algoritmo originale del 1998.

Conclusioni

L'algoritmo del 1998, nonostante fosse incompleto, non si comporta male rispetto a quello completo. Il software nel tempo si è trasformato migliorando e migrando sulla maggior numero di sistemi possibile, grazie alla riscrittura del codice, ora interamente in C/C++. Inizialmente funzionava solo in ambiente MSDOS e Windows 3.11. Ora può funzionare ovunque esista un compilatore C/C++. Nel 2019 si è evoluto ma è divenuto purtroppo più dispendioso. Se lo si vorrà utilizzare bisognerà raggiungere un compromesso tra la compressione ottenuta e le risorse utilizzate per ottenerla poiché non tutti i sistemi potrebbero eseguirlo. In alternativa si può continuare a usare l'algoritmo del 1998 che richiede relativamente poche risorse ed è meno esigente, soprattutto in termini di memoria utilizzata.

Ringraziamenti

Volevamo ringraziare ancora una volta tutta la Segreteria di Redazione del CEN che si è dimostrata molto veloce ed efficiente. Un ringraziamento particolare è dovuto alla dott.ssa Rossella Celi per la sua cordialità, la disponibilità e la professionalità. Ringraziamo infine il revisore per l'accuratezza, la precisione e la professionalità adottata durante la revisione. Grazie.

Bibliografia

Del Negro C., Di Bella A., Ferrucci F., Napoli R., Sicali A., (1998*). Automated System for Magnetic Surveillance of Active Volcanoes.* In Final Report Tekvolc, Technique and Method Innovation in Geophysical Research, Monitoring and Early Warning at Active Volcanoes. Commission of European Communities Environment Programme, Contract ENV4 CT95 0251.

Del Negro C., Di Bella A., Napoli R., Sicali A., (1999*). Development of an automated console prototype for control of remote magnetic sensors.* In Interim Report Tomave, electromagnetic and potential field integrated tomographies applied to volcanic environments. Commission of European Communities Environment Programme, Contract ENV4 CT98 0697.

Del Negro C., Napoli R., Sicali A., (2002). *Automated system for magnetic monitoring of active volcanoes*. Bull. Volcanol., 64, 94-99.

Elias P., (1975). *Universal Codeword Sets and Representations of the Integers*. IEEE TRANS. ON INF. THEO., vol. IT-21, no. 2, March 1975 (1975-03-01), pages 194 - 203, XP000946246. doi:10.1109/TIT.1975.1055349

Fano R.M., (1949). *The transmission of information*. Technical Report No. 65. Cambridge (Mass.), USA, Research Laboratory of Electronics at MIT.

Huffman D.A., (1952). *A Method for the Construction of Minimum Redundancy Codes*. Proceedings of the IRE, 40, 1098-1101.

Shannon C. E., (1948). *A Mathematical Theory of Communication*. Bell System Technical Journal, vol. 27, 379–423 (luglio), 623–656 (ottobre).

Sicali A., (2000). *Entropia dei segnali continui nel tempo*. Computer Programming n. 89. marzo 2000, 68-72. ISSN 1123-8526.

Sicali A., Amantia A., Cappuccio P., (2018). *Realizzazione di un sistema operativo Client-Server per la gestione di stazioni geomagnetiche remote e relativo simulatore in ambiente Unix*. Rapporti tecnici INGV n°. 395. ISSN 2039-7941

Sicali A., Amantia A., Cappuccio P., (2018). *Evoluzione ventennale (1998-2018) del sistema Mag-Net per l'acquisizione dei segnali dalla rete magnetica dell'Etna e dell'isola di Stromboli*. Rapporti tecnici INGV n°. 403. ISSN 2039-7941

Sicali A., Amantia A., Cappuccio P., (2020). *Tecnologie dei regolatori solari standalone per il monitoraggio magnetico dell'Etna e dell'isola di Stromboli*. Rapporti tecnici INGV n°. 414. ISSN 2039-7941

APPENDICE A1

Appendice A1 – Sorgenti del sistema di compressione ace.h

```
/*
  Sistema di compressione ACEAS studiato per la versione network
  del sistema MagNet
  Antonino Sicali (c) 1998-2019
  Istituto Nazionale di Geofisica e Vulcanologia
  Catania
*/


#ifndef _ACE_H
#define _ACE_H

#ifdef _ACE_EVOLUTION
#define NWORD           2
#else
#define NWORD           6
#define NDELTA          20
#endif

#define DECIMAL_BASE 10

#ifdef _ACE_EVOLUTION

#define MAX_PARAMETER_SIZE 20
#define MAX_LINES          10000

#define HEADER_TIME_SIZE (sizeof(short)*6)

typedef struct EVOLUTION_FIELD {
    char v[MAX_PARAMETER_SIZE];
    short int n;
    short int left;
    short int right;
    unsigned char lbit;
    unsigned long hd;
    unsigned short bin[NWORD];
} EVOLUTION_FIELD;

#endif // _ACE_EVOLUTION

// Dichiarazione delle funzione ASSEMBLER utilizzate.
// Le chiamate alle funzioni sono del tipo C.
#ifdef _ASM_SOURCE
extern "C" {
        void  COMPTABTIME(unsigned  char  far  *header_time,unsigned  char  far
*tabella,unsigned short int lmisura);
        void  DECTABTIME(unsigned  char  far  *header_time,unsigned  char  far
*tabella,unsigned short int lmisura);
        void  INITFIRSTTEXTTIME(unsigned  char  far  *text,unsigned  short  far
*header_time);
        void  INITFIRSTHEADERTIME(unsigned  char  far  *text,unsigned  short  far
*header_time);
#else
#define far
#endif
    #ifndef _ACE_EVOLUTION
    void COMPTAB(unsigned short *header,unsigned char far *tabella,unsigned short int
lmisura);
    void DECTAB(unsigned short *header,unsigned char far *tabella,unsigned short int
lmisura);
    short int LENG(unsigned char far *tabella,unsigned short int lmisura);
    void COMPRESS(unsigned char far *tabella,unsigned short far *uscita,unsigned char
far *misura,unsigned short int lmisura);
        void  DECOMPRESS(unsigned  char  far  *tabella,unsigned  short  far
*ingresso,unsigned char far *misura,unsigned short int lmisura);
        short int GET(unsigned short far *source,unsigned short far *number,short
int n,short int nbyte);
        #else
        int COMPRESS(EVOLUTION_FIELD *tree,short int n,unsigned char *misura,unsigned
short int lmisura);
        short int GET(EVOLUTION_FIELD *tree,unsigned char *misura,unsigned short int
lmisura,unsigned short far *source,short int nbyte);
        void CONVERTTIMETOBINARY(unsigned short *t,unsigned long *number);
        void CONVERTTIMETONUMBER(unsigned short *t,unsigned long *number);
```

```
    #endif

        #ifndef _ACE_EVOLUTION
        void  FORMATHEADERTIME(unsigned  short  far  *binary1,unsigned  char  far
*text2,unsigned char far *header_time);
        void   FORMATHEADER(unsigned   short   far   *header,unsigned   char   far
*misura,unsigned short int lmisura);
        #else
        void FORMATHEADER(EVOLUTION_FIELD *header,unsigned short &n,unsigned char
far *misura,unsigned short int lmisura);
        void  FORMATHEADERTIME(unsigned  short  far  *binary1,unsigned  char  far
*text2,EVOLUTION_FIELD *header_time,unsigned short &n);
        #endif
        void RESMISURA(unsigned char far *misura,unsigned char *string,unsigned short
leng,unsigned short n);
        short int STORE(unsigned short far *drain,unsigned short far *number,short
intn);
        #ifndef _ACE_EVOLUTION
        unsigned  char  GETNDELTA(unsigned  short  far  *binary1,unsigned  char  far
*text2,unsigned char far *header_time);
        void    GETTIME(unsigned    short    far    *binary1,unsigned    char    far
*text2,EVOLUTION_FIELD *header_time,unsigned char ndelta);
        #else
        unsigned    char    GETNDELTA(unsigned    short    *binary1,unsigned    char
*text2,EVOLUTION_FIELD *header_time,unsigned short n);
        void GETTIME(unsigned short *binary1,unsigned char *text2,unsigned short
*delta);
        #endif
        void  CALCDELTAMEASURE(unsigned  char  far  *measure1,unsigned  char  far
*measure2,unsigned   char   far   *delta,unsigned   short   n,unsigned   char   far
*nodigit,unsigned char far *base);
        void   CALCMEASURE(unsigned   char   far   *measure1,unsigned   char   far
*delta,unsigned   char   far   *measure2,unsigned   short   n,unsigned   char   far
*nodigit,unsigned char far *base);
        void CONVERTMEASURETONUMBER(unsigned char far *measure,unsigned short far
*number,unsigned short n);
        void CONVERTNUMBERTOMEASURE(unsigned short far *number,unsigned char far
*measure,unsigned short n);

        unsigned long MISCRC(unsigned char far *misura,unsigned short leng);
        unsigned long OLD_MISCRC(unsigned char far *misura,unsigned short leng);

    unsigned short GETYEAR(unsigned char far *misura);
    unsigned short GETMONTH(unsigned char far *misura);
    unsigned short GETDAY(unsigned char far *misura);
    unsigned short GETHOUR(unsigned char far *misura);
    unsigned short GETMIN(unsigned char far *misura);
    unsigned short GETSEC(unsigned char far *misura);

    #ifndef _ACE_EVOLUTION
        unsigned char far * GETPOINTER(unsigned short int i,unsigned short int
flag,unsigned char far *base,unsigned char far *number);
    #endif //_ACE_EVOLUTION

        unsigned   short   TESTMISURA(unsigned   char   far   *misura,unsigned   char
*string,unsigned short leng,unsigned short n);
#ifdef _ASM_SOURCE
            }
#else


#ifndef _ACE_EVOLUTION
void TABTIME(unsigned char far * header,unsigned char far *tabella,unsigned short int
lmisura);

#define DECTABTIME TABTIME
#define COMPTABTIME TABTIME
#endif // _ACE_EVOLUTION

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                          ;;
//;  La funzione INITFIRSTHEADERTIME copia il valore temporale               ;;
//;  iniziale nell'header temporale.                                         ;;
//;                                                                          ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
#define INITFIRSTHEADERTIME(a,b) FORMATBINARY(b,a);
#define INITFIRSTTEXTTIME(a,b) FORMATTEXT(a,b)

unsigned short GETVALUE(unsigned char *s,int ndigit);
void SETVALUE(unsigned char *s,int ndigit,unsigned short v);
void GETDELTA(unsigned short *delta,unsigned char *binary_header,unsigned char
ndelta);
void ADDTIME(unsigned short *binary,unsigned short *delta);
void FORMATTEXT(unsigned char far *text2,unsigned short far *binary1);
void FORMATBINARY(unsigned short *binary,unsigned char *text);
void SUBTIME(unsigned char *delta,unsigned short far *binary1);
#ifdef _ACE_EVOLUTION
unsigned short SCANHEADERTIME(unsigned short *binary,EVOLUTION_FIELD
*header_time,unsigned short n);
void INSERTHEADER(EVOLUTION_FIELD *header_time,unsigned short &n,unsigned short
*binary);
#else
unsigned char SCANHEADERTIME(unsigned char *delta,unsigned char far *header_time);
void INSERTHEADER(unsigned char far *header_time,unsigned short *delta);
#endif
void DIVSHORT(unsigned short *a,unsigned short b);
void MULSHORT(unsigned short *a,unsigned short b);
void SUBLARGE(unsigned short *a,unsigned short *b);
void ADDLARGE(unsigned short *a,unsigned short *b);
unsigned short LOG2(unsigned short *binary);
short int GETNDAY(unsigned short year,unsigned char month);

#define HOUR_SIZE  2
#define MIN_SIZE   2
#define SEC_SIZE   2
#define DAY_SIZE   2
#define MONTH_SIZE 2

#endif

#endif //_ACE_H
```

APPENDICE A2

Appendice A2 - Sorgenti del sistema di compressione ace.cpp

```
/*
  Sistema di compressione ACEAS studiato per la versione network
  del sistema MagNet
  Antonino Sicali (c) 1998-2019
  Istituto Nazionale di Geofisica e Vulcanologia
  Catania
*/


//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                  Chiamate di funzioni ASSEMBLER                    ;;
//;                              in C/C++                              ;;
//;                                                                    ;;
//; Le funzioni ASSEMBLER possono essere richiamate                    ;;
//; in C/C++, seguendo alcune regole generali:                         ;;
//; 1) Le funzioni in C sono dichiarate in ASSEMBLER ponendo un        ;;
//;    underscore davanti al nome: _function in ASSEMBLER viene        ;;
//;    chiamata dal C come function.                                   ;;
//; 2) I parametri nello stack vengono inseriti da destra verso        ;;
//;    sinistra così che gli offset dello stack aumenteranno           ;;
//;    verso destra ed i parametri aggiuntivi si dovranno              ;;
//;    aggiungere a destra per non modificare ogni volta il codice.;;
//; 3) Lo stack viene formato e disfatto dal compilatore C/C++         ;;
//;    l'unica cosa da fare sarà ripulire lo stack dalle proprie       ;;
//;    variabili e ritornare con una ret per i modelli near e retf ;;
//;    per i modelli far.                                              ;;
//; 4) per riferirsi alle variabili dello stack la prassi è di         ;;
//;    utilizzare il registro BP con un indice aggiuntivo. Per         ;;
//;    i modelli near il primo parametro a sinistra disterà 4 byte ;;
//;    per i modelli far disterà 6 byte. Nel caso delle DWORD          ;;
//;    verrà inserita nello stack prima la parte alta e poi quella ;;
//;    bassa.                                                          ;;
//; 5) I valori di 16 bit ritornati dalla funzioni risiederanno nel;;
//;    registro AX mentre i valori di 32 bit saranno memorizzati       ;;
//;    nella coppia di registri DX:AX.                                 ;;
//; Per comprendere il codice non è necessario sapere nulla di più.;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

// ID_TABELLA definisce le tabelle per il
// segnale di cui hanno bisogno alcune funzioni
// come COMPRESS, LENG e DECOMPRESS
// ID_HEADERVALUE definisce l'header di ciascun
// segnale, ID_MEASURE definisce il valore
// iniziale del segnale

// Queste costanti vengono utilizzate per la costruzione
// della stringa di definizione

#ifndef _CODE_
#include "ace/ace.evolution.h"
#include "ace/aceas.evolution.h"
#else
#include "ace.evolution.h"
#include "aceas.evolution.h"
#endif

#define MAXVALUE     255
#define MAXYEAR     10000

#define SCRC   1024       //  Questa  costante  definisce  la  lunghezza  del  buffer
CRCBUFFER

// Queste variabili vengono utilizzate dalle varie funzioni
// per memorizzare valori temporaneamente

unsigned short SW=0;                  //  Utilizzata  dalla  STORE  memorizza  la  word
correntemente processata
unsigned short BIT_S=0;         // Utilizzata dalla STORE memorizza il numero di bit
validi in _SW
unsigned short GR=0;          // Utilizzata dalla GET memorizza la word correntemente
processata
unsigned short BIT_R=0;          // Utilizzata dalla GET memorizza il numero di bit
```

```
validi in _GR
unsigned short A[NWORD];       // Utilizzate per operazioni sui
unsigned short B[NWORD];       // numeri lunghi
unsigned  short  C[HEADER_TIME_SIZE/sizeof(short)];  //  Utilizzata  per  operazioni
temporali
unsigned char CB[16];          // Utilizzata per operazioni temporali
unsigned char CRCBUFFER[SCRC];     // Buffer utilizzato per la verifica delle misure
elaborate
unsigned short YEAR_OFFSET=0;  // Mantiene l'offset a cui trovare l'anno
unsigned short YEAR_LENGTH=0;  // Mantiene la lunghezza della stringa rappresentante
l'anno: 2 o 4 caratteri
unsigned short DAY_OFFSET=0;   // Mantiene l'offset a cui trovare il  giorno
unsigned short MONTH_OFFSET=0; // Mantiene l'offset a cui trovare il mese
unsigned short HOUR_OFFSET=0;  // Mantiene l'offset a cui trovare l'ora
unsigned short MIN_OFFSET=0;   // Mantiene l'offset a cui trovare i minuti
unsigned short SEC_OFFSET=0;   // Mantiene l'offset a cui trovare i secondi
// Variabili locali di sola lettura per l'elaborazione
// del segnale temporale
unsigned char SUP[]={MAXVALUE,12,0,24,60,60};
unsigned char DAYS[]={31,0,31,30,31,30,31,31,30,31,30,31};

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
//;  La funzione TABTIME provvede durante la comp/decompressione      ;;
//;  a ricostruire una tabella per la conversione tra delta time      ;;
//;  ed indice memorizzato nella word totale.                         ;;
//;                                                                   ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

void TABTIME(unsigned char far * header,unsigned char far *tabella,unsigned short int
lmisura)
{
        unsigned char n=header[HEADER_TIME_SIZE];
        int i=lmisura*(DECIMAL_BASE+1);
        tabella[i++]=n;
        unsigned char c=0;
        int k;
        for (k=0;k<n;k++) { tabella[i++]=c; c++; }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
//;  La funzione GETDELTA estrae dall'header delle variazioni         ;;
//;  temporali un delta. La funzione riceve un puntatore far          ;;
//;  (DS:BX) all'header e un indice di riferimento.                   ;;
//;  Il delta time sarà memorizzato nella struttura binaria           ;;
//;  puntata da ES:DI.                                                ;;
//;                                                                   ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
void GETDELTA(unsigned  short  *delta,EVOLUTION_FIELD  *binary_header,unsigned  char
ndelta)
{
        memcpy(delta,binary_header[ndelta].v,HEADER_TIME_SIZE);
}
#else
void  GETDELTA(unsigned  short  *delta,unsigned  char  *binary_header,unsigned  char
ndelta)
{

memcpy(delta,binary_header+(HEADER_TIME_SIZE+1)+ndelta*HEADER_TIME_SIZE,HEADER_TIME_
SIZE);
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
//;  La funzione GETMON ritorna il numero di giorni del mese          ;;
//;  contenuto nella struttura puntata da ES:DI. Questa funzione      ;;
//;  se richiamata da SUBTIME riceve in BP il valore 1 mentre se      ;;
//;  richiamata da ADDTIME riceve il valore 0. Il numero di giorni ;;
//;  sarà ritornato in DL                                             ;;
//;                                                                   ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETMON(unsigned short *binary,BOOL sub)
```

```
{
    unsigned short month=binary[1];
    unsigned short year=binary[0];

    if (sub) {
        month—;
        if (!month) { month=12; year—; }
    }

    return GETNDAY(year,month);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione GETSUP ritorna il limite superiore di              ;;
//;  ogni membro della struttura temporale puntata da ES:DI.        ;;
//;  La funzione è sfruttata durante l'esecuzione di differenze     ;;
//;  e somme temporali con riporto. Il limite sarà ritornato in DL.;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETSUP(unsigned short *binary,int i,BOOL sub)
{
    unsigned short v=SUP[i];

    if (!v) v=GETMON(binary,sub);
        else if (v == MAXVALUE) v=MAXYEAR;

    return v;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione SUBTIME esegue la differenza temporale con riporto;;
//;  tra le strutture temporali puntate da ES:DI e DS:SI.           ;;
//;  Il risulato dell'operazione ES:DI-DS:SI verrà memorizzato in   ;;
//;  ES:DI mentre il valore precedentemente puntato da ES:DI verrà ;;
//;  copiato in DS:SI per un'utilizzazione futura.                  ;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void SUBTIME(unsigned short *binary1,unsigned short *binary2)
{
    short i;
    unsigned short c=0,v,bk;

    // La struttura temporale è lunga sei byte
        // ed i campi disposti da sinistra verso destra
        // comprendono anno, mese, giorno, ore, minuti e secondi.

    for (i=5;i>=0;i—) {
            binary2[i]+=c;
            bk=binary1[i];
            if (binary1[i] < binary2[i]) {
                v=GETSUP(binary1,i,TRUE);
                binary1[i]+=v;
                c=1;
            }
            else c=0;
            binary1[i] -= binary2[i];
            binary2[i] = bk;
        }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione ADDMONTHS esegue una pre-somma (solo mese ed anno);;
//;  per eliminare un errore di riporto sul giorno che non possiede;;
//;  un limite superiore costante. Il puntatore DS:SI indica il     ;;
//;  delta time mentre ES:DI indica la struttura temporale          ;;
//;  iniziale.                                                      ;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void ADDMONTHS(unsigned short *binary1,unsigned short *binary2)
{
        // Il numero variabili di giorni
        // per ogni mese implica che durante
```

```
        // il riporto si debba conoscere il
        // mese e l'anno finale. Questa funzione
        // esegue una pre-somma sul mese e sull'anno
        binary1[1] += binary2[1];
        unsigned short v=GETSUP(binary1,1,FALSE);
        if (binary1[1] > v) {
            binary1[1] -= v;
            binary1[0]++;
        }
    binary1[0] += binary2[0];

        binary2[0]=0;
        binary2[1]=0;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione ADDTIME esegue la somma temporale con riporto       ;;
//;  tra le strutture temporali puntate da ES:DI e DS:SI.            ;;
//;  La seconda struttura rappresenta il delta time.                 ;;
//;  Il risulato dell'operazione ES:DI+DS:SI verrà memorizzato in   ;;
//;  ES:DI.                                                          ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void ADDTIME(unsigned short *binary,unsigned short *delta)
{
    unsigned short c=0;
    unsigned short v,t;
    short i;

        ADDMONTHS(binary,delta); // Effettua una pre-somma dei campi mensili e
annuali

        for (i=5;i>=0;i—) {
        binary[i] += c+delta[i];
        v=GETSUP(binary,i,FALSE); // Acquisisce il limite superire e genera eventuale
riporto
        if (i == 1 || i == 2) t=1; else t=0;
        if (binary[i] >= v+t) { binary[i] -= v; c=1; } else c=0;
        }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione GETVALUE converte un valore numerico ASCII          ;;
//;  composto da due cifre in un valore binario. I due caratteri     ;;
//;  sono puntati dal puntatore far DS:BX mentre il valore binario ;;
//;  è ritornato nel registro AX.                                    ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETVALUE(unsigned char *s,int ndigit)
{
    unsigned short v=0;
    unsigned short f=1;
    short i;

    for (i=ndigit-1;i>=0;i—) {
            v += s[i]*f;
            f *= DECIMAL_BASE;
    }

    return v;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione GETNDAY ritorna in CL il numero di giorni           ;;
//;  del mese passato in CL ed in CH l'anno. Ritorna il numero       ;;
//;  di giorni del mese corrente in cx.                              ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
short int GETNDAY(unsigned short year,unsigned char month)
{
        short int d;
```

```
        d=DAYS[month-1];
        if (!d) { if (year % 4) d=28; else d=28; }
        return d;
}




//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione FORMATBINARY converte un valore temporale ASCII   ;;
//;  puntato da DS:BX in binario, memorizzandolo nella struttura   ;;
//;  temporale puntata da ES:DI.                                   ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void FORMATBINARY(unsigned short *binary,unsigned char *text)
{
        binary[0]=GETVALUE(text+YEAR_OFFSET,YEAR_LENGTH);
    binary[1]=GETVALUE(text+MONTH_OFFSET,MONTH_SIZE);
    binary[2]=GETVALUE(text+DAY_OFFSET,DAY_SIZE);
    binary[3]=GETVALUE(text+HOUR_OFFSET,HOUR_SIZE);
    binary[4]=GETVALUE(text+MIN_OFFSET,MIN_SIZE);
    binary[5]=GETVALUE(text+SEC_OFFSET,SEC_SIZE);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione SETVALUE converte un valore numerico binario      ;;
//;  contenuto in AX in un valore numero ASCII. I due caratteri    ;;
//;  saranno memorizzati nelle locazioni puntate da DS:BX.         ;;
//;  è ritornato nel registro AL.                                  ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void SETVALUE(unsigned char *s,int ndigit,unsigned short v)
{
    short i;

    for (i=ndigit-1;i>=0;i—) {
            s[i]=v%DECIMAL_BASE;
            v /= DECIMAL_BASE;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione FORMATTEXT converte un valore temporale binario   ;;
//;  contenuto nella struttura puntata da ES:DI in ASCII,          ;;
//;  memorizzandolo all'indirizzo puntato da DS:BX                 ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void FORMATTEXT(unsigned char *text,unsigned short *binary)
{

    SETVALUE(text+YEAR_OFFSET,YEAR_LENGTH,binary[0]);
    SETVALUE(text+MONTH_OFFSET,MONTH_SIZE,binary[1]);
    SETVALUE(text+DAY_OFFSET,DAY_SIZE,binary[2]);
    SETVALUE(text+HOUR_OFFSET,HOUR_SIZE,binary[3]);
    SETVALUE(text+MIN_OFFSET,MIN_SIZE,binary[4]);
    SETVALUE(text+SEC_OFFSET,SEC_SIZE,binary[5]);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione GETTIME estrae il valore temporale associato alla ;;
//;  misura.                                                       ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
void GETTIME(unsigned short far *binary1,unsigned char far *text2,unsigned short far
*delta)
{
        ADDTIME(binary1,delta); // Somma il delta time alla struttura binaria
        FORMATTEXT(text2,binary1); // Converte la struttura binaria in stringa ASCII
}
#else
```

```
void GETTIME(unsigned short far *binary1,unsigned char far *text2,unsigned char far
*header_time,unsigned char ndelta)
{
    GETDELTA(C,header_time,ndelta); // Acquisisce il delta time utilizzando l'indice
         ADDTIME(binary1,C); // Somma il delta time alla struttura binaria
         FORMATTEXT(text2,binary1); // Converte la struttura binaria in stringa ASCII
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                     ;;
//;  La funzione SCANHEADETIME converte un delta time puntato         ;;
//;  da ES:DI in un indice ritornato in AL. Per operare la            ;;
//;  conversione la funzione ha bisogno di un puntatore (DS:BX)       ;;
//;  all' header contenente tutte le variazioni temporali possibili.;;
//;                                                                     ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
unsigned    short    SCANHEADERTIME(unsigned    short    *binary,EVOLUTION_FIELD
*header_time,unsigned short n)
{
    unsigned short i;

    for (i=0;i<n;i++) if (!memcmp(binary,header_time[i].v,HEADER_TIME_SIZE)) break;

    return i;
}
#else
unsigned short SCANHEADERTIME(unsigned short *binary,unsigned char *header_time)
{
    unsigned short i;
    header_time += HEADER_TIME_SIZE;
    unsigned short n=header_time[0]; // Copia il numero delle variazioni possibili
    header_time++; // Punta alla parte dell'headertime dedicata
                   // all'archiviazione delle variazioni temporali.

    for (i=0;i<n;i++) {
        if (!memcmp(binary,header_time,HEADER_TIME_SIZE)) break;
        header_time+=HEADER_TIME_SIZE;
    }
    // Questa funzione ritornerà sempre un indice
         // valido poichè verrà eseguita dopo la scansione
         // iniziale del segnale temporale
    return i;
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETNDELTA ritorna un indice di riferimento alla      ;;
//;  variazione temporale occorsa.                                     ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
unsigned char GETNDELTA(unsigned short *binary1,unsigned char *text2,EVOLUTION_FIELD
*header_time,unsigned short n)
#else
unsigned char GETNDELTA(unsigned short far *binary1,unsigned char far *text2,unsigned
char far *header_time)
#endif
{
        FORMATBINARY(C,text2);    // Converte il valore temporale ASCII
        SUBTIME(C,binary1); // Calcola il delta time
        return SCANHEADERTIME(C,header_time
                       #ifdef _ACE_EVOLUTION
                       ,n
                       #endif
                       ); // Ricerca nell'headertime il delta
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione INSERTHEADER inserisce una variazione temporale      ;;
//;  nell'header temporale sempre che non sia già presente. Il        ;;
//;  valore del delta time è puntato da ES:DI mentre header da        ;;
```

```
//;  DS:BX.                                                            ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
void INSERTHEADER(EVOLUTION_FIELD *header_time,unsigned short &n,unsigned short *binary)
{
    unsigned short i;

    for (i=0;i<n;i++) if (!memcmp(binary,header_time[i].v,HEADER_TIME_SIZE)) break;
    if (i < n) { header_time[i].n++; return; }
         memcpy(header_time[i].v,binary,HEADER_TIME_SIZE);
         header_time[i].n=1;
         n++;
}
#else
void INSERTHEADER(unsigned char *header_time,unsigned short *binary)
{
    header_time += HEADER_TIME_SIZE;
    unsigned char *p=(unsigned char*) header_time;
    unsigned short n=p[0];
    unsigned short i;

    p++;
    for (i=0;i<n;i++) {
        if (!memcmp(binary,p,HEADER_TIME_SIZE)) break;
        p+=HEADER_TIME_SIZE;
    }
    if (i < n) return;
         memcpy(p,binary,HEADER_TIME_SIZE);
         header_time[0]++;
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione FORMATHEADERTIME opera una classificazione delle      ;;
//;  variazioni temporali di ogni misura.                              ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
void FORMATHEADERTIME(unsigned short far *binary1,unsigned char far *text2,EVOLUTION_FIELD *header_time,unsigned short &n)
#else
void FORMATHEADERTIME(unsigned short far *binary1,unsigned char far *text2,unsigned char far *header_time)
#endif
{
    FORMATBINARY(C,text2); // Converte il valore temporale ASCII in binary
         SUBTIME(C,binary1); // Calcola il delta time corrente  e
         INSERTHEADER(header_time
                #ifdef _ACE_EVOLUTION
                ,n
                #endif
              ,C);       // lo inserisce nell'headertime
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione CONVERTNUMBERTOMEASURE converte un numero binario ;;
//;  in ASCII.                                                         ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void CONVERTNUMBERTOMEASURE(unsigned short far *number,unsigned char far *measure,unsigned short n)
{
         short i;

    for (i=n-1;i>=0;i—) {
        memcpy(B,number,NWORD*sizeof(unsigned short));
        DIVSHORT(number,DECIMAL_BASE);     // Scompone il numero in cifre attraverso
        MULSHORT(number,DECIMAL_BASE);     // divisioni e moltiplicazioni successive
        SUBLARGE(B,number);
        measure[i]=B[0];
        DIVSHORT(number,DECIMAL_BASE);
```

```
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione CONVERTMEASURETONUMBER converte un numero ASCII    ;;
//;  in binario.                                                     ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void  CONVERTMEASURETONUMBER(unsigned  char  far  *measure,unsigned  short  far
*number,unsigned short n)
{
    short i;
    memset(A,0,sizeof(unsigned short)*NWORD); // Inizializza il buffer A ad 1
    A[0]=1;
         memset(number,0,sizeof(unsigned short)*NWORD);

    for (i=n-1;i>=0;i—) {
        memcpy(B,A,sizeof(unsigned short)*NWORD);
        MULSHORT(B,measure[i]);   // Moltiplica la cifra per il multiplo di dieci
presente in B e
        ADDLARGE(number,B);           // somma il risultato al totale
        MULSHORT(A,DECIMAL_BASE);        // Moltiplica per dieci il multiplo presente
in A
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione CONVERTTIMETONUMBER compatta un tempo in binario   ;;
//;  nel formato YYYY/MM/DD HH:MM:SS                                 ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void CONVERTTIMETONUMBER(unsigned short *t,unsigned long *number)
{
   *number=t[0];
    *number *= 12;
    *number += t[1];
    *number *= 31;
    *number += t[2];
    *number *= 24;
    *number += t[3];
    *number *= 60;
    *number += t[4];
    *number *= 60;
    *number += t[5];
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione CONVERTTIMETOBINARY divide un tempo compattato     ;;
//;  in binario nel formato YYYY/MM/DD HH:MM:SS                      ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void CONVERTTIMETOBINARY(unsigned short *t,unsigned long *number)
{
    t[5]=*number % 60;
    *number /= 60;
    t[4]=*number % 60;
    *number /= 60;
    t[3]=*number % 24;
    *number /= 24;
    t[2]=*number % 31;
    *number /= 31;
    t[1]=*number % 12;
    *number /= 12;
    t[0]=*number;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione CALCDELTAMEASURE calcola la derivata del segnale   ;;
//;  in ingresso.                                                    ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
void   CALCDELTAMEASURE(unsigned   char   far   *measure1,unsigned   char   far
*measure2,unsigned   char   far   *delta,unsigned   short   n,unsigned   char   far
*nodigit,unsigned char far *base)
{
    unsigned char sign=0;
    unsigned char *tmp;
    unsigned char s;
    unsigned char c=0;
    short i,j;
    unsigned char v,u;

    memset(delta,0,n+1);
    for (i=0;i<n;i++) {
            v=measure1[i];
            u=measure2[i];
            if (v < u) break;
           if (v > u) { sign=1; tmp=measure1; measure1=measure2; measure2=tmp; break;
}
    }

    delta[0]=sign;

    for (i=n-1;i>=0;i–) {
        j=(sign?measure2:measure1)-base+i;
        v=nodigit[0];
        if (v != 0xff && v == j) { nodigit–; continue; }
        s=measure2[i];
        s -= c;
       if (s == 0xff) { c=1; s=9; } else if (s < measure1[i]) { c=1; s += DECIMAL_BASE;
} else c=0;
        s -= measure1[i];

        delta[i+1]=s;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione CALCMEASURE ricostruisce il segnale dalla derivata;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void CALCMEASURE(unsigned char far *measure1,unsigned char far *delta,unsigned char
far *measure2,unsigned short n,unsigned char far *nodigit,unsigned char far *base)
{
    unsigned char sign=delta[0];
    unsigned char c=0;
    char s;
    short i,j;
    unsigned char v;

    for (i=n-1;i>=0;i–) {
        j=measure1-base+i;
        v=nodigit[0];
        if (v != 0xff && v == j) {
            nodigit–;
            continue;
        }
        s=measure1[i];
        if (sign) { // subdelta
            s -= c+delta[i+1];
            c=0;
            if (s < 0)  { s += DECIMAL_BASE; c=1; }
        }
        else
        {
            s += c+delta[i+1];
            c=0;
            if (s > 9) { s -= DECIMAL_BASE; c=1; }
        }
        measure2[i] = s;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                     ;;
```

```
//;  La funzione TESTLAST esegue un test numerico sulla stringa     ;;
//;  di misura e un test sul range dei singoli campi temporali.     ;;
//;  La stringa di misura ripulita di constanti e di simboli        ;;
//;  non numerici viene puntata da ES:BX. La lunghezza della        ;;
//;  stringa è indicata dal registro CX mentre AX ritorna il        ;;
//;  risultato del test. Se AX è nullo la misura contiene errori.   ;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BOOL TESTLAST(unsigned char *measure,int n)
{
    unsigned short i;
    unsigned short v;

    for (i=0;i<n;i++) {
        if (measure[i] > '9') return 0;
        if (measure[i] < '0') return 0;
        measure[i] -= '0';
    }

    unsigned short month=GETVALUE(measure+MONTH_OFFSET,2);
    if (month < 1 || month > 12) return 0;

    unsigned short year=GETVALUE(measure+YEAR_OFFSET,YEAR_LENGTH);

    unsigned short day=GETVALUE(measure+DAY_OFFSET,2);
    if (day < 1 || day > GETNDAY(year,month)) return 0;

    v=GETVALUE(measure+HOUR_OFFSET,2);
    if (v > 23) return 0;
    v=GETVALUE(measure+MIN_OFFSET,2);
    if (v > 59) return 0;
    v=GETVALUE(measure+SEC_OFFSET,2);
    if (v > 59) return 0;

        return 1;
}

BOOL  T_IDML_COST(unsigned  char  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
    unsigned char j=string[0];
    unsigned char v=string[1];
    unsigned short i;

    for (i=0;i<n;i+=leng) {
            if (misura[i+j] != v) return FALSE;
           misura[i+j]='0';
    }

    return 1;
}

BOOL  T_IDML_RANGE(unsigned  char  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
        unsigned char j=string[0];
    unsigned char b=string[1];
    unsigned char a=string[2];
    unsigned short i;

    for (i=0;i<n;i+=leng) if (misura[i+j] < a || misura[i+j] > b) return FALSE;

    return TRUE;
}

void  T_IDML_SUB(unsigned  char  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
    unsigned char v=string[1];
    unsigned char j=string[0];
    unsigned short i;

    for (i=0;i<n;i+=leng) {
        misura[i+j] -= v;
        misura[i+j] += '0';
```

```
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                  ;;
//;  La funzione TESTMISURA esegue un test sulla stringa             ;;
//;  di misura basandosi sulla stringa di definizione.               ;;
//;  Ritorna zero se ci sono errori nella stringa di misura.         ;;
//;                                                                  ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short TESTMISURA(unsigned char far *misura,unsigned char *string,unsigned
short leng,unsigned short n)
{
        unsigned short i=0;
        unsigned short j;

        while(string[i] != IDML_END) {
            switch(string[i]) {
                case IDML_COST:
                    if (!T_IDML_COST(misura,string+i+1,leng,n)) {
                            return 0;
                    }
                    i+=2;
                    break;
                case IDML_RANGE:
                    if (!T_IDML_RANGE(misura,string+i+1,leng,n)) {
                            return 0;
                    }
                    i+=3;
                    break;
                case IDML_SIZE:
                case IDML_NODIGIT:
                    i++;
                    break;
                case IDML_VALUE:
                    i+=3;
                    break;
                case IDML_SUB:
                    T_IDML_SUB(misura,string+i+1,leng,n);
                    i+=2;
                    break;
                case IDML_TIME:
                    i+=7;
                    break;
            }
        i++;
        }

    for (j=0;j<n;j+=leng) if (!TESTLAST(misura+j,leng)) {
            return 0;
    }

    return 1;
}

void  R_IDML_COST(unsigned  char  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
    unsigned char j=string[0];
    unsigned char v=string[1];
    unsigned short i;

    for (i=0;i<n;i+=leng) misura[i+j]=v;
}

void  R_IDML_SUB(unsigned  char  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
    unsigned char v=string[1];
    unsigned char j=string[0];
    unsigned short i;

    for (i=0;i<n;i+=leng) {
        misura[i+j] -= '0';
        misura[i+j] += v;
```

```
    }
}




//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione RESMISURA esegue la conversione da cifre numeriche;;
//;  a cifre ASCII della misura, basandosi sulla stringa di         ;;
//;  definizione.                                                   ;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void  RESMISURA(unsigned  char  far  *misura,unsigned  char  *string,unsigned  short
leng,unsigned short n)
{
    unsigned short i;

    for (i=0;i<n;i++) misura[i] += '0';

    i=0;
    while(string[i] != IDML_END) {
            switch(string[i]) {
                case IDML_COST:
                    R_IDML_COST(misura,string+i+1,leng,n);
                    i+=2;
                    break;
                case IDML_VALUE:
                case IDML_RANGE:
                    i+=3;
                    break;
                case IDML_SUB:
                    R_IDML_SUB(misura,string+i+1,leng,n);
                    i+=2;
                    break;
                case IDML_SIZE:
                case IDML_NODIGIT:
                    i++;
                    break;
                case IDML_TIME:
                    i+=7;
                    break;
            }
            i++;
    }
}



//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione MISCRC calcola un checksum della stringa di misura;;
//;  Il valore ritornato è lungo 32 bit.                            ;;
//;                                                                 ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned long MISCRC(unsigned char far *misura,unsigned short leng)
{
        unsigned long crc=0L;
        unsigned short v;
        unsigned short  i;

        for (i=0;i<leng;i++) {
            v = misura[i];
            v += v << 8;
            crc += v;
        }

        return crc;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                 ;;
//;  La funzione GETPOINTER ritorna un puntatore alla tabella o     ;;
//;  header specificati.                                            ;;
```

```
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef _ACE_EVOLUTION
unsigned char far *GETPOINTER(unsigned short int i,unsigned short int flag,unsigned
char far *base,unsigned char far *number)
{
    unsigned short j;

        if (!i) switch(flag) {
                    case ID_HEADERTIME:   // Indica la richiesta del puntatore
all'headertime.
                return base;
              case ID_HEADERVALUE: // Indica il puntatore all'headervalue.
                    // Il numero tra parentesi indica sei byte della struttura
binaria contenente
                     // il numero di variazioni, e sei byte per ogni singola
variazione.
                return base+ HEADER_TIME_SIZE+1+(NDELTA*HEADER_TIME_SIZE);
              case ID_TABELLA:
                    // Lunghezza dell'headertime.
                                                                    return    base
+(HEADER_TIME_SIZE+1+(NDELTA*HEADER_TIME_SIZE))+(number[0]+1)*2;
                // Legge la lunghezza della stringa    numerica e vi aggiunge 1
per coprire il segno della derivata.
            case ID_MEASURE: // Indica un puntatore al valore iniziale del segnale
selezionato.
                                                                    return    base    +
(HEADER_TIME_SIZE+1+(NDELTA*HEADER_TIME_SIZE)+NDELTA+1)+(number[0]+1)*(HEADER_TIME_S
IZE+1);  // Lunghezza dell'headertime e della tabella di conversione della variazione
                     // temporale. Legge la lunghezza della stringa numerica e vi
aggiunge 1 per coprire il segno.
                // Aggiunge undici byte per ogni cella della tabella e due byte
per l'headervalue
                }

    unsigned short k=0;
    for (j=0;j<i;j++) k += number[j]+1;

    k *= 13;
    k += (NWORD*2)*j;

        switch(flag) {
        case ID_TABELLA:
            k += (number[j]+1)*2;
            break;
        case ID_MEASURE:
            k += (number[j]+1)*13;
            break;
        }

        return base+k+HEADER_TIME_SIZE+1+(NDELTA*HEADER_TIME_SIZE)+NDELTA+1;
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione ADDLARGE esegue una somma con riporto tra due         ;;
//;  numeri lunghi specificati da ES:DI e DS:BX.                       ;;
//;  Il risultato viene trascritto in ES:DI.                           ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void ADDLARGE(unsigned short *a,unsigned short *b)
{
    unsigned short i;
    unsigned short c=0;
    unsigned short v;

    for (i=0;i<NWORD;i++) {
        v=a[i];
        a[i] += b[i]+c;
        c=v > a[i]?1:0;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
//;                                                                    ;;
//;  La funzione SUBLARGE esegue una differenza con riporto tra due;;
//;  numeri lunghi specificati da ES:DI e DS:BX (ES:DI-DS:BX).     ;;
//;  Il risultato viene trascritto in ES:DI.                         ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void SUBLARGE(unsigned short *a,unsigned short *b)
{
    unsigned short i,v;
    unsigned short c=0;

    for (i=0;i<NWORD;i++) {
        v=a[i];
        a[i] -= b[i]+c;
        c=v < a[i]?1:0;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione DIVSHORT esegue una divisione tra un                ;;
//;  numero lungo ed un numero di 16 bit. Il numero lungo è puntato;;
//;  da DS:BX mentre il numero corto è passato attraverso il         ;;
//;  registro DX. Il risultato è puntato da DS:BX.                   ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void DIVSHORT(unsigned short *a,unsigned short b)
{
    unsigned long v;
    short i;

    v = a[NWORD-1];
    for (i=NWORD-2;i>=0;i—) {
                v <<= 16;
                v |= a[i];
                a[i] = v / b;
                a[i+1]=(v/b)>>16;
                if (i > 0) v %= b;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione MULSHORT esegue una moltiplicazione tra un          ;;
//;  numero lungo ed un numero di 16 bit.                            ;;
//;  Il numero lungo è puntato                                       ;;
//;  da DS:BX mentre il numero corto è passato attraverso il         ;;
//;  registro DX. Il risultato è puntato da DS:BX.                   ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void MULSHORT(unsigned short *a,unsigned short b)
{
    unsigned long v;
    unsigned short c=0;
    unsigned short i;

    for (i=0;i<NWORD;i++) {
        v = a[i];
        v *= b;
        v += c;
        a[i] = v &0x0000FFFF;
        c = v >> 16;
    }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione COMPRESS si occupa della compressione del segnale.;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifdef _ACE_EVOLUTION
int COMPRESS(EVOLUTION_FIELD *tree,short int n,unsigned char *misura,unsigned short
int lmisura)
{
    int i;
```

```
    for (i=0;i<n;i++) if (!memcmp(tree[i].v,misura,lmisura)) break;
    return i;
}
#else
void COMPRESS(unsigned char far *tabella,unsigned short far *uscita,unsigned char far
*misura,unsigned short int lmisura)
{
    unsigned short b,n,i;
    unsigned char v;

         memset(uscita,0,sizeof(unsigned short)*NWORD);
         memset(A,0,sizeof(unsigned short)*NWORD);
         A[0]=1;

         for (i=0;i<lmisura;i++) {
           v=misura[i]+1; // La cifra viene incrementata a causa della presenza
dell'indice iniziale nella tabella.
         n=tabella[0]; // Viene letto l'indice della tabella per sapere quante cifre
nella colonna sono state identificate.
          // Se ci sono meno di una cifra la quantità di informazione è nulla e la
colonna non avrà alcun peso nella
            // codifica binaria. Il puntatore alla tabella si sposta sulla cifra
dell'intervallo compatto per un'imminente lettura.
         if (n <= 1) { tabella += (DECIMAL_BASE+1); continue; }
         memcpy(B,A,sizeof(unsigned short)*NWORD);           // Copia il numero lungo
da A a B.
         MULSHORT(A,n);
         b=tabella[v];
         MULSHORT(B,b); // Moltiplica il numero presente in B per la cifra compatta e
aggiunge il risultato
         ADDLARGE(uscita,B);        // al numero lungo presente nel buffer di uscita
         tabella+=DECIMAL_BASE+1;
         }       // Processa le altre cifre.
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                     ;;
//;  La funzione LOG2 estrae il logaritmo base 2 del numero lungo       ;;
//;  puntato da DS:BX e memorizza il risultato nel registro DX.         ;;
//;                                                                     ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short LOG2(unsigned short *binary)
{
    unsigned short r=16*NWORD;
    unsigned short v;
    short i,j;

    for (i=NWORD-1;i>=0;i–) {
        v=binary[i];
        if (!i && v < 2) return 0;
        for (j=0;j<16;j++) {
            if (v & 0x8000) return r;
            v <<= 1;
            r–;
        }
    }

    return 0;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                     ;;
//;  La funzione DECOMPRESS si occupa della decompressione              ;;
//;  del segnale.                                                       ;;
//;                                                                     ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef _ACE_EVOLUTION
void DECOMPRESS(unsigned char far *tabella,unsigned short far *ingresso,unsigned char
far *misura,unsigned short lmisura)
{
    unsigned short n,i;

         for (i=0;i<lmisura;i++) {
```

```
        n=tabella[0];
        if (n == 1) misura[i]=tabella[n];
        if (!n || n == 1) { tabella += DECIMAL_BASE+1; continue; }
        memcpy(A,ingresso,sizeof(unsigned short)*NWORD);
        DIVSHORT(ingresso,n);     // Divide e moltiplica la codeword per l'indice
        MULSHORT(ingresso,n);     // estratto dalla tabella per isolare la cifra
dell'intervallo compatto attraverso una
        SUBLARGE(A,ingresso);   // sottrazione con il valore contenuto in A.
        misura[i]=tabella[A[0]+1];
        DIVSHORT(ingresso,n);
        tabella += DECIMAL_BASE+1;
         }
}
#endif




//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
//;  La funzione FORMATHEADER costruisce l'header per la              ;;
//;  funzione COMPRESS.                                               ;;
//;                                                                   ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef _ACE_EVOLUTION
void FORMATHEADER(unsigned short far *header,unsigned char far *misura,unsigned short
int lmisura)
{
        unsigned short i;

        for (i=0;i<lmisura;i++) header[i] |= 1 << misura[i];
}
#else
void   FORMATHEADER(EVOLUTION_FIELD   *header,unsigned   short   &n,unsigned   char
*misura,unsigned short int lmisura)
{
        int i;

    for (i=0;i<n;i++) if (!memcmp(header[i].v,misura,lmisura)) break;
    if (i < n) { header[i].n++; return; }
    memcpy(header[i].v,misura,lmisura);
    header[i].n=1;
    n++;
}
#endif

#ifndef _ACE_EVOLUTION
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
//;  La funzione COMTAB si occupa di creare dall'header del segnale;;
//;  una tabella per l'accesso diretto e di facile utilizzazione     ;;
//;  per la funzione COMPRESS.                                        ;;
//;                                                                   ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void COMPTAB(unsigned short far *header,unsigned char far *tabella,unsigned short int
lmisura)
{
        unsigned short i,j;
        unsigned char k;
        unsigned short v;

        for (i=0;i<lmisura;i++) {
            v=header[i];
            k=0;
            for (j=0;j<DECIMAL_BASE;j++) {
                if (v&0x0001) { tabella[j+1]=k; k++; }
                v >>= 1;
            }
            tabella[0]=k;
            tabella += DECIMAL_BASE+1;
        }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                   ;;
```

```
//;  La funzione DECTAB si occupa di creare dall'header del segnale;;
//;  una tabella per l'accesso diretto e di facile utilizzazione   ;;
//;  per la funzione DECOMPRESS.                                   ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
void DECTAB(unsigned short far *header,unsigned char far *tabella,unsigned short int
lmisura)
{
        unsigned short i,v,j;
        unsigned char k;

        for (i=0;i<lmisura;i++) {
            v=header[i];
            k=0;
            for (j=0;j<DECIMAL_BASE;j++) {
                if (v&0x0001) { k++; tabella[k]=j; }
                v >>= 1;
            }
            tabella[0]=k;
            tabella += DECIMAL_BASE+1;
        }
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione LENG calcola la lunghezza della codeword di       ;;
//;  memorizzazione del segnale. Il valore ritornato è lungo 16 bit;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
short int LENG(unsigned char far *tabella,unsigned short int lmisura)
{
    unsigned short i;
    unsigned short k;

    memset(A,0,sizeof(unsigned short)*NWORD); // Inizializza il buffer A ad 1
    A[0]=1;
        for (i=0;i<lmisura;i++) {
        k=tabella[i*(DECIMAL_BASE+1)];
        if (k) MULSHORT(A,k);
        }

        // Calcola il numero di bit necessari per
        // memorizzare la misura in forma compatta.
    return LOG2(A);
}
#endif

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                ;;
//;  La funzione STORE si occupa di ridurre l'arrotondamento della ;;
//;  compressione convertendo le codeword in bit e memorizzandoli  ;;
//;  uno di seguito all'altro. Il valore ritornato indica i byte   ;;
//;  presenti ancora nel buffer di ingresso.                       ;;
//;                                                                ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
short int STORE(unsigned short far *drain,unsigned short far *number,short int n)
{
        int r=0;
        unsigned short k,bits,i,v;


        if (!number) { // Se number uguale a zero significa che i buffer si debbono
svuotare.
             if (!BIT_S) return 0; // Se _BIT_S è uguale a zero non ci sono bit da
memorizzare e la funzione ritorna.
            // Richiama la word corrente memorizzata nella variabile statica _SW
            // Calcola il numero di bit necessari per
            // completare la word e la memorizza nel buffer di uscita.
            *drain = SW >> (15-BIT_S);
            return 2;
        }

        k=0;
        i=0;
        for (bits=0;bits<n;bits++) {
```

```
            if (!k) { k=16; v=number[i]; i++; }
            if (v&0x0001) SW |= 0x8000;
            v >>= 1;
            k--;
            BIT_S++;
            if (BIT_S == 16) {
                *drain = SW;
                drain++;
                BIT_S=0;
                SW=0;
                r+=2;
            } else SW >>= 1;
        }

        return r;
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GET si occupa di estrarre dalla successione di        ;;
//;  bit creata con la funzione STORE le codeword necessarie per la;;
//;  decompressione. Il valore ritornato indica il numero di byte      ;;
//;  presenti ancora nel buffer di ingresso.                           ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef _ACE_EVOLUTION
short int GET(unsigned short far *source,unsigned short far *number,short int n,short
int nbyte)
#else
short  int  GET(EVOLUTION_FIELD  *tree,unsigned  char  *misura,unsigned  short  int
lmisura,unsigned short far *source,short int nbyte)
#endif
{
    unsigned short i=0;

    #ifndef _ACE_EVOLUTION
    unsigned short v,k,bits;

        memset(number,0,sizeof(unsigned short)*NWORD);
        k=0;
        v=0;
        #else
        int index=0;
        #endif

        #ifdef _ACE_EVOLUTION
        while (true) {
    #else
    for (bits=0;bits<n;bits++) {
    #endif
        if (!BIT_R) {
                if (i == nbyte) return nbyte+1; // Ritorna il numero di byte passati+1
per indicare un errore nel file.
                GR=source[i];
                i++;
                BIT_R=16;
        }
        if (GR&0x0001)
                        #ifdef _ACE_EVOLUTION
                        index=tree[index].right; else index=tree[index].left;
                        #else
                        v |= 0x8000;
                        #endif
        GR >>= 1;
        BIT_R--;

        #ifdef _ACE_EVOLUTION
        if (tree[index].right < 0) {
                memcpy(misura,tree[index].v,lmisura);
                break;
        }
        #else
        k++;
        if (k == 16) {
            *number=v;
```

```
                number++;
                k=0;
                v=0;
            } else v >>= 1;
            #endif
             }

             #ifndef _ACE_EVOLUTION
        v >>= 15-k;
        *number=v;
        #endif
             return nbyte-i*sizeof(unsigned short);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETDAY estrae il giorno dalla misura passata come ;;
//;  argomento.                                                        ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 unsigned short GETDAY(unsigned char far *misura)
 {
    unsigned short i;

    memcpy(CB,misura+DAY_OFFSET,DAY_SIZE);
    for (i=0;i<DAY_SIZE;i++) CB[i] -= '0';
    return GETVALUE(CB,DAY_SIZE);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETMONTH estrae il giorno dalla misura passata    ;;
//;  come argomento.                                                   ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 unsigned short GETMONTH(unsigned char far *misura)
 {
    unsigned short i;

    memcpy(CB,misura+MONTH_OFFSET,MONTH_SIZE);
    for (i=0;i<MONTH_SIZE;i++) CB[i] -= '0';
    return GETVALUE(CB,MONTH_SIZE);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETYEAR estrae il giorno dalla misura passata     ;;
//;  come argomento.                                                   ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 unsigned short GETYEAR(unsigned char far *misura)
 {
    unsigned short i;

    memcpy(CB,misura+YEAR_OFFSET,YEAR_LENGTH);
    for (i=0;i<YEAR_LENGTH;i++) CB[i] -= '0';
    return GETVALUE(CB,YEAR_LENGTH);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETHOUR estrae il giorno dalla misura passata     ;;
//;  come argomento.                                                   ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETHOUR(unsigned char far *misura)
{
    unsigned short i;

    memcpy(CB,misura+HOUR_OFFSET,HOUR_SIZE);
    for (i=0;i<HOUR_SIZE;i++) CB[i] -= '0';
    return GETVALUE(CB,HOUR_SIZE);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
//;                                                                    ;;
//;  La funzione GETMIN estrae il giorno dalla misura passata come    ;;
//;  argomento.                                                        ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETMIN(unsigned char far *misura)
{
    unsigned short i;

    memcpy(CB,misura+MIN_OFFSET,MIN_SIZE);
    for (i=0;i<MIN_SIZE;i++) CB[i] -= '0';
    return GETVALUE(CB,MIN_SIZE);
}

//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//;                                                                    ;;
//;  La funzione GETSEC estrae il giorno dalla misura passata come     ;;
//;  argomento.                                                        ;;
//;                                                                    ;;
//;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
unsigned short GETSEC(unsigned char far *misura)
{
    unsigned short i;

    memcpy(CB,misura+SEC_OFFSET,SEC_SIZE);
    for (i=0;i<SEC_SIZE;i++) CB[i] -= '0';
    return GETVALUE(CB,SEC_SIZE);
}
```

APPENDICE A3

Appendice A3 – Sorgenti del sistema di compressione aceas.h

```
/*
  Sistema di compressione ACEAS studiato per la versione network
  del sistema MagNet
  Antonino Sicali (c) 1998-2019
  Istituto Nazionale di Geofisica e Vulcanologia
  Catania
*/


#ifndef _ACEAS_H
#define _ACEAS_H
#include <string.h>
#include <stdio.h>
#if defined(_WINDOWS__) || defined(_WIN32)
#include <windows.h>
#else
#ifndef _LINUX
#include <dos.h>
#endif
#endif
//#define _CODE_
#ifndef _CODE_
#include "config.h"
#include "ace/ace.evolution.h"
#else  //_CODE_
#include "ace.evolution.h"
#include "../config.h"
#endif //_CODE_

#define DATE_SEPARE '-'
#define TIME_SEPARE ':'

// Queste costanti permettono di acquisire il puntatore ad un header
// o ad una tabella.
// Ogni modifica si dovrà duplicare anche nel file ace.asm
#define ID_HEADERTIME   0
#define ID_TABELLA      1
#define ID_HEADERVALUE  2
#define ID_MEASURE      3

// Costanti utilizzabili nella stringa di definizione.
// Ogni modifica si dovrà duplicare anche nel file ace.asm
#define IDML_SUB     0
#define IDML_COST            1
#define IDML_VALUE      2
#define IDML_SIZE       3
#define IDML_RANGE      4
#define IDML_NODIGIT    5
#define IDML_TOTAL      6
#define IDML_DELAY      7
#define IDML_TIME       8
#define IDML_END        255

// Signature dell'algoritmo
#ifndef _ACE_EVOLUTION
#define SIGNATURE "ACEAS Ver. 4.00 Mobile"
#else
#define SIGNATURE "ACEAS v.5.00 Evolution"
#endif

#define ACEEXT     "ACE"

#define NOMEASURE  3
#define CONTINUE   4
#define NOACEFILE  5

#define TEST        6

#define ENABLE_ACEAS_DEBUG

#ifndef  ENABLE_ACEAS_DEBUG
#define ACEAS_DEBUG_HANDLE 0
#else
```

```
#define ACEAS_DEBUG_HANDLE aceas_debug_handle
#endif

#define ACE_TMP_FILE "ace.tmp"

#define MAX_VALUES 50

extern unsigned short SW,BIT_S,GR,BIT_R;
extern unsigned short YEAR_OFFSET,YEAR_LENGTH,MONTH_OFFSET,DAY_OFFSET;
extern unsigned short HOUR_OFFSET,MIN_OFFSET,SEC_OFFSET;

// Struttura di intestazione del file compresso dalla release 4.
typedef struct HEADER_ACEAS{
                    // Segnatura del file compresso.
                    char signature[32];
                   // Checksum delle misure.
                    unsigned long crc;
                   // Numero di misure compresse.
                    unsigned long nmisure;
                   // Lunghezza della stringa di definizione.
                    unsigned short leng_ml;
                    unsigned long error;
                    //unsigned long nfile;
                    //unsigned long nmisfile;
                    } HEADER_ACEAS;

class Aceas {
        #ifdef  ENABLE_ACEAS_DEBUG
        int aceas_debug_handle;
        #endif
             int i_test,n_test,q_test;
             #ifndef _ACE_EVOLUTION
             unsigned char  *table_value_test[MAX_VALUES];
             unsigned char  *table_time_test;
             unsigned char *header_time_test;
        unsigned short *header_value_test[MAX_VALUES];
        unsigned short t_test;
             #else
             unsigned short header_time_test[HEADER_TIME_SIZE/sizeof(short)];
        EVOLUTION_FIELD *header_value_test[MAX_VALUES];
             #endif

             BOOL end_file_test;
             unsigned short j_test,k_test;
             HEADER_ACEAS header_test;

             #ifndef _ACE_EVOLUTION
             unsigned char *header_time;
             unsigned char *table_value[MAX_VALUES];
        unsigned char *table_time;
        unsigned short *header_value[MAX_VALUES];
        #else
        unsigned short header_time[HEADER_TIME_SIZE/sizeof(short)];
        EVOLUTION_FIELD *header_value[MAX_VALUES];
        unsigned short header_measures[MAX_VALUES][NWORD];
        #endif

             // Memorizza la lunghezza delle stringhe numeriche
                     // rappresentanti il segnale all'interno di ciascuna
             // misura.
             unsigned char list[100];
        // Memorizza gli offset delle stringhe numeriche.
             unsigned char offset[100];
             // Memorizza il tipo di codifica adottata per ciascun
        // segnale.
             unsigned char type[100];
        // Memorizza l'offset di tutti i caratteri IDML_NODIGIT.
             unsigned char nodigit[100];
        // Memorizza il numero di caratteri IDML_NODIGIT.
             unsigned short n_nodigit;
        // Memorizza la lunghezza della codeword di ciascun segnale.
             short lbit[100];
             unsigned short nlist;
             #ifndef _ACE_EVOLUTION
        // Memorizza il numero di stringhe numeriche trovate.
```

```
                unsigned char far *Headers;
                // Memorizza il puntatore base per gli header e tabelle.
                unsigned short SHeaders;
                #endif


               // Specifica la lunghezza del buffer temporaneo.
                enum {SWAPSIZE=10000}; //10000
                // Memorizza l'intestazione del file compresso.
                HEADER_ACEAS header_aceas;
               // Memorizzano puntatori a buffer temporanei.
                char *temp_buffer;
                char *wr;
               // Memorizza il tempo base per le operazioni temporali.
                unsigned short binary1[6];
               // Memorizza l'indice temporale.
                char delta;
                // Memorizza la misura o la derivata in maniera temporanea.
                unsigned char delta_measure[1024];
               // Memorizza la misura base per le operazioni sui segnali.
                unsigned char previous[1024];
               // Memorizza gli handle a file di ingresso e uscita.
                short source,drain,tmp;
               // Memorizza i puntatori per il buffer di ingresso/uscita.
              unsigned char *bufferin;
                unsigned char *bufferout;
                // Memorizza il puntatore di scrittuta sul buffer temporale.
                unsigned short pwrite;
                // Memorizza lo stato di inizzializzazione della scansione.
                BOOL initscan;
                // Memorizza lo stato di inizzializzazione della compressione.
                BOOL initcomp;

                void WriteToBuffer(unsigned short handle,unsigned char *buffer,unsigned
short size,BOOL comp,BOOL last=FALSE);
                void InitHeaders(void);
                void InitScan(unsigned char *buffer);
                bool InitComp(unsigned char *buffer);
                void FreeHeader(void);
                void SaveHeaders(unsigned short handle);
                void LoadHeaders(unsigned short handle
                            #ifdef _ACE_EVOLUTION
                            ,unsigned short *header_time
                            #endif
                            );
        public:
                // Memorizza la stringa di definizione.
                unsigned char String[1024];
                // Memorizza la lunghezza della stringa di misura.
                unsigned short bytemisura;

                Aceas(void);
                ~Aceas(void) {
               // Libera la memoria allocata.
               #ifndef _ACE_EVOLUTION
               if (Headers != 0) FreeHeaders();
               #endif
                             if (bufferin != 0) delete[](bufferin);
                             if (bufferout != 0) delete[](bufferout);
                              }
                BOOL Init(unsigned char* filename=NULL);
                void FreeHeaders(void);
                BOOL AllocHeaders(unsigned char *string);
                void OpenComp(unsigned char *filename);
                void CloseComp(void);
                BOOL Scan(unsigned char *buffer,unsigned short size);
                unsigned short Compress(unsigned char *buffer,unsigned short size);
                BOOL Decompress(char *sourcefile,char *drainfile);
                BOOL Compress(char *sourcefile,char *drainfile,unsigned char *string);
                unsigned long GetError(void) { return header_aceas.error; }
                BOOL OpenTest(char *sourcefile);
                BOOL ScanTest(void);
                BOOL CloseTest(void);
                };
```

```
unsigned char *parser(unsigned char *string,char *measure);
void print(FILE *handle,unsigned char *string);
short int GetList(unsigned char *string,char *list,char *offset,unsigned char
*type,unsigned short &size,unsigned char *nodigit,unsigned short &n_nodigit);
void digitalizer(char *string,char *measure,unsigned char *list,unsigned char
*offset,unsigned short nlist,unsigned short bytemisura,unsigned long *data);
short int Getbytemisura(unsigned char *string);
void Addbytemisura(unsigned char *string,int idml,int v1,int v);
int Strlen(unsigned char *buffer);
void Strcpy(unsigned char *drain,unsigned char *source);
void GetTime(unsigned char *misura,struct date &d,struct time &t);
void reorder(unsigned char *string);

#endif
```

APPENDICE A4

Appendice A4 – Sorgenti del sistema di compressione aceas.cpp

```
/*
  Sistema di compressione ACEAS studiato per la versione network
  del sistema MagNet
  Antonino Sicali (c) 1998-2019
  Istituto Nazionale di Geofisica e Vulcanologia
  Catania
*/


/*
  Il programma mira a realizzare tre obiettivi:
    1. Calcolo della variazione temporale associata a ciascuna misura
    2. Calcolo della derivata sui segnali avente un IDML_DELAY come
       attributo.
    3. Applicazione della funzione antispike.
ogni obiettivo non è raggiunto da una sola funzione ed, in linea di
massima, ad ogni funzione ne corrisponde una complementare utilizzata
durante la decodifica del file. Il codice sorgente della classe è
riportato in quattro file differenti: aceas.cpp, aceas.h, ace.asm ace.h.
Del primo obiettivo si occupano le funzioni: CONVERT, RESMISURA, GETTIME,
GETNDELTA, FORMATHEADERTIME, INITFIRSTHEADERTIME, INITFIRSTTEXTTIME,
COMPTABTIME, DECTABTIME, MISCRC, GETPOINTER, TESTMISURA.
Ciascuna funzione adempie ad un compito specifico e tutte insieme
riescono a descrivere in maniera esauriente un qualunque segnale temporale.
Di seguito sono riportati i passi che portano alla compressione ed alla
ricostruzione del segnale temporale:
    Compressione.
    1. Estrazione del segnale temporale iniziale dalla prima stringa di
       misura (INITFIRSTHEADERTIME).
    2. Scansione di ciascuna misura successiva alla prima e calcolo del
       delta time (FORMATHEADERTIME).
    3. Compilazione tabella temporale (COMPTABTIME).
    4. Associa ciascun delta time ad un indice variabile tra zero e
       NDELTA, registrandolo all'inizio del primo segnale (GETNDELTA).
    Decodifica.
    1. Compila tabella (DECTABTIME).
    2. Estrae dal primo segnale l'indice associato alla variazione
       temporale.
    3. Estrae il delta time dall'headertime e lo somma al segnale
       temporale corrente per ottenere il segnale temporale successivo
       (GETTIME).
    4. Ricostruisce tutti i caratteri costanti (RESMISURA).
  Il secondo obiettivo è raggiunto attraverso le funzioni: CONVERT, RESMISURA,
CALCDELTAMEASURE, CALCMEASURE, CONVERTMEASURETONUMBER, CONVERTNUMBERTOMEASURE,
MISCRC, GETPOINTER, TESTMISURA.
Di seguito sono riportati i passi necessari al programma per operare una
codifica/decodifica del segnale del tipo IDML_DELAY:
    Codifica.
    1. Memorizza il valore iniziale del segnale (CONVERTMEASURETONUMBER).
    2. Calcolo della derivata (CALCDELTAMEASURE).
    3. Sostituire la derivata al segnale originale e operare una codifica
       di tipo IDML_TOTAL.
    Decodifica.
    1. Richiama dall'headermeasure il valore iniziale di ciascun segnale
       (CONVERTNUMBERTOMEASURE).
    2. Estrae la derivata dalla codeword (DECOMPRESS).
    3. Somma o sottrae il valore della derivata al segnale corrente
       ottenendo quello successivo (CALCMEASURE).
  Il terzo obiettivo è raggiunto attraverso le funzioni: COMPRESS, DECOMPRESS,
FORMATHEADER, COMTAB, DECTAB, CONVERT, RESMISURA, LENG, STORE, GET, MISCRC,
GETPOINTER, TESTMISURA. La funzione antispike applicata dalle funzioni elencate
sopra permette di ridurre l'influenza delle variazioni infinitesimali
sull'occupazione del segnale e quindi sulla lunghezza della codeword. Tale
funzione si preoccupa di comprimere gli intervalli derivati dalla divisione di
ciascuna stringa numerica in cifre. Si consideri una tabella di valori:

                                43689.19
                                43689.31
                                43689.47
                                  43689.51
                                43689.62
                                43689.68
```

ogni stringa numerica può essere divisa in colonne larghe una cifra ottenendo dei sottoinsiemi contenenti numeri da zero a nove. Se le stringhe numeriche in esame rappresentano un segnale si possono individuare un numero di cifre su ogni colonna che diminuisce verso sinistra:

(1) (1) (1) (1) (1) (punto) (5) (5)

L'entropia associata a ciascun gruppo diminuisce al diminuire del numero cifre, così che le colonne più a destra possederanno maggior informazione rispetto a quelle più a sinistra. Il contenuto di informazione è rivelato dal programma compattando ciascun insieme e scrivendo ogni cifra attraverso un indice variabile tra zero e il numero massimo di cifre rilevate. Per esempio la colonna più a destra dell'esempio precedente può essere riscritta dal programma come:

(4) (0) (2) (0) (1) (3)
9 1 7 1 2 8

considerando che ci sono cinque cifre e vale la corrispondenza:

1=0, 2=1, 7=2, 8=3, 9=4

Si può osservare che una variazione infinitesimale per quanto ampia può causare una variazione di cifre minima nel modello a colonne influenzando minimamente la lunghezza della codeword di memorizzazione. Di seguito sono illustrate le fasi che portano alla codifica/decodifica del segnale da parte della funzione antispike:

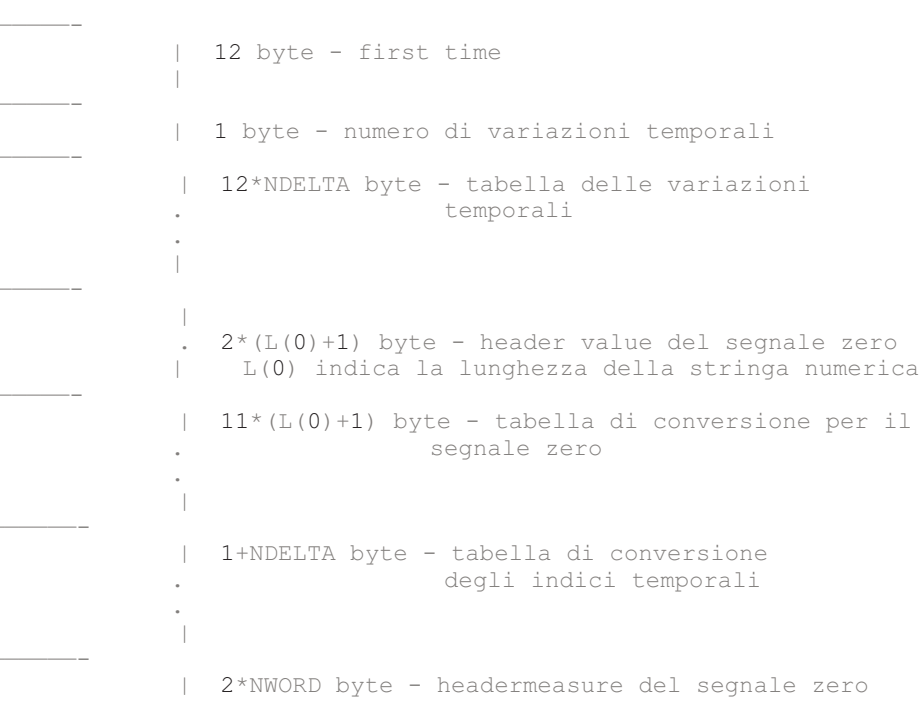
Codifica.

1. Scansione delle colonne per l'identificazione delle cifre (FORMATHEADER).
2. Compilazione tabella (COMPTAB).
3. Conversione tra indici non compatti e compatti riferiti all'headervalue di ciascuna colonna (COMPRESS).
4. Trascrizione della codeword sotto forma di serie di bit (STORE).

Decodifica.

1. Compilazione tabella (DECTAB).
2. Estrazione della codeword dalla serie bit in ingresso (GET).
3. Espansione dell'insieme attraverso la conversione da indice compatto riferito all'headervalue in indici non compatti (DECOMPRESS).
4. Ricostruzione dei caratteri costanti ed in generale di tutte le parti della misura (RESMISURA).

Un ruolo importante ricopre la funzione GETPOINTER utilizzata per richiedere puntatori alle varie tabelle e header necessari alla codifica/decodifica binaria. GETPOINTER riceve un puntatore base ed una serie di costanti che analizza per ricostruire la posizione di una tabella o di un header secondo lo schema:



```
 _______
.             .  2*(L(i)+1) byte - header value del segnale di ordine
.             .                 'i'. L(i) indica la lunghezza della
 .............                  stringa numerica del segnale.
.             .  11*(L(i)+1) byte - tabella di conversione per il
.             .                   segnale di ordine 'i'.
 .............
.             .  2*NWORD byte - headermeasure per il segnale di
 .            .                  ordine 'i'.
  .............
.             .
 .            .
```

La misura può contenere più segnali differenti ed ognuno verrà trattato indipendetemente dagli altri. Durante la codifica/decodifica verrà calcolato un checksum sulla misura per evitare errori nell'interpretazione dei dati. Durante la codifica verrà anche effettuato un test sulle misure per escludere tutte quelle che non rispecchiano lo standard contenuto nella stringa di definizione.

```
*/

//#define _CODE_

#ifndef _LINUX
#include <conio.h>
#include <io.h>
#include <process.h>
#include <mem.h>
#else
#include <unistd.h>
#include <errno.h>
#endif
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifndef _CODE_
#include "config.h"
#include "magnet2k.h"
#include "ace/aceas.evolution.h"
#include "ace/ace.evolution.h"
#include "timer.h"
#else // _CODE_
#include "aceas.evolution.h"
#include "ace.evolution.h"
#endif //_CODE_
#include <fcntl.h>
#include <sys/stat.h>

#ifdef _WIN32
#include "../timer.h"
#endif // _WIN32

#ifdef _ACE_EVOLUTION

#define  MAX_SORT_LEVELS  50000

EVOLUTION_FIELD evolution_buffer[MAX_VALUES][MAX_LINES];
unsigned short nevolution_buffer[MAX_VALUES];

int begin_sort[MAX_SORT_LEVELS];
int end_sort[MAX_SORT_LEVELS];

bool quickSort(EVOLUTION_FIELD *data,int n)
{
      int i=0,left,right;
      EVOLUTION_FIELD pivot;

      begin_sort[0]=0;
      end_sort[0]=n;

      while (i>=0) {
            left=begin_sort[i];
            right=end_sort[i]-1;
```

```
            if (left<right) {
               pivot=data[left];
               if (i == MAX_SORT_LEVELS-1) return false;
               while (left<right) {
               while (data[right].v <= pivot.v && left < right) right--;
               if (left < right) data[left++]=data[right];
               while (data[left].v >= pivot.v && left < right) left++;
               if (left < right) data[right--]=data[left]; }
               data[left]=pivot;
               begin_sort[i+1]=left+1;
               end_sort[i+1]=end_sort[i];
               end_sort[i++]=left;
             }
             else { i--; }
      }

      return true;
}

void huffmanFill(EVOLUTION_FIELD *data,int i,unsigned long hd,int lbit)
{
    if (data[i].left < 0) {
            data[i].lbit=lbit;
            data[i].hd=hd>>(32-lbit);
            return;
    }
    huffmanFill(data,data[i].left,hd>>1,lbit+1);
    huffmanFill(data,data[i].right,(hd>>1)|0x80000000,lbit+1);
}

void huffmanCompact(EVOLUTION_FIELD *data,unsigned short &n)
{
    int k=0,i;

    for(i=0;i<n;i++) {
        if (data[i].left >= 0) continue;
        data[k]=data[i];
        k++;
    }
    n=k;
}

void huffman(EVOLUTION_FIELD *data,unsigned short &n)
{
    int i,last;
    EVOLUTION_FIELD tmp;

    for(i=0;i<n;i++) { data[i].right=data[i].left=-1; }

    if (n == 1) return;

    last=n;

    while(n > 1) {
        data[last]=data[n-1];
        data[n-1]=data[n-2];
        data[n-2].n += data[n-1].n;
        data[n-2].left=last;
        data[n-2].right=n-1;
        n--;
        last++;
        for (i=n-2;i>=0;i--) {
                    if (data[i].n >= data[i+1].n) break;
                    tmp=data[i];
                    data[i]=data[i+1];
                    data[i+1]=tmp;
        }
    }
    n=last;
}

#endif // _ACE_EVOLUTION

extern void writeLog(unsigned char *s);
```

```
int Strlen(unsigned char *buffer)
{
        int i=0;
        while(buffer[i] != IDML_END) i++;
        return i;
}

void Strcpy(unsigned char *drain,unsigned char *source)
{
        int i=0;
        while(source[i] != IDML_END) { drain[i]=source[i]; i++; }
        drain[i]=IDML_END;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione OpenComp si occupa di inizializzare le variabili //
//  e di aprire il file di uscita.                                //
//                                                                //
////////////////////////////////////////////////////////////////////
void Aceas::OpenComp(unsigned char *filename)
{
    #ifdef _ACE_EVOLUTION
    memset(evolution_buffer,0,sizeof(evolution_buffer));
    memset(nevolution_buffer,0,sizeof(nevolution_buffer));
    memset(header_value_test,0,sizeof(header_value_test));
    memset(header_time_test,0,sizeof(header_time_test));
    memset(header_measures,0,sizeof(header_measures));
    #else
    memset(Headers,0,SHeaders);
    #endif
        memset(temp_buffer,0,bytemisura);
        memset(wr,0,bytemisura);
        pwrite=0;                       // Inizializza le varibili
        header_aceas.crc=0;             // usate durante la
        header_aceas.nmisure=0;         // compressione.
        header_aceas.error=0;
        initscan=FALSE;
        initcomp=FALSE;
        if (drain) return;              // Ritorna se il file è già stato aperto
        #ifndef _LINUX
        _fmode=O_BINARY;
        #endif
        drain=open((char*)filename, O_CREAT|O_WRONLY|O_TRUNC
            #ifdef _LINUX
            ,S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
            #endif
            );
        //drain=creat(,S_IWRITE); // Crea il file di uscita
        char s[128];

sprintf(s,"%s%c%s%c%s",PROGRAMMDIRECTORY,DIR_CHAR,TMP_DIRECTORY,DIR_CHAR,ACE_TMP_FIL
E);
        tmp=open(s, O_CREAT|O_RDWR|O_TRUNC
          #ifdef _LINUX
          ,S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
          #endif
          );
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione WriteToBuffer si occupa di memorizzare              //
//  temporaneamente i dati in uscita in un buffer per scriverli   //
//  tutti contemporaneamente in un secondo tempo                  //
//                                                                //
////////////////////////////////////////////////////////////////////
void Aceas::WriteToBuffer(unsigned short handle,unsigned char *buffer,unsigned short
size,BOOL comp,BOOL last)
{
        if (pwrite+size >= SWAPSIZE) {
                                        // Ricostruisce la misura ASCII dalla
misura binaria leggendo
                                           // le informazioni dalla stringa di
definizione.
```

```
                                        if (!comp) {

RESMISURA(bufferout,String,bytemisura,pwrite);
                                                header_aceas.crc                +=
MISCRC(bufferout,pwrite);
                                                  }
                                        // Scrive l'intero contenuto del
                                        // buffer temporaneo sul disco.
                                        if                                (handle)
write(handle,bufferout,pwrite);
                                        // Inizializza il puntatore di
                                        // scrittura.
                                        pwrite=0;
                                    }
        memcpy(bufferout+pwrite,buffer,size); // Copia di dati dal buffer
        pwrite += size;                       // di ingresso al buffer
                                                // temporaneo ed avanza
                                                // il puntato di scrittura.
        if (last == TRUE) {
                                //  Ricostruisce  la  misura  ASCII  dalla  misura
binaria leggendo
                                // le informazioni dalla stringa di definizione.
                                if (!comp) {

RESMISURA(bufferout,String,bytemisura,pwrite);
                                        header_aceas.crc                        +=
MISCRC(bufferout,pwrite);
                                         }
                                // Scrive il contenuto del buffer
                                // temporaneo sul disco se la chiamata
                                // alla funzione è l'ultima.
                                if (handle) write(handle,bufferout,pwrite);
                                // Inizializza il puntatore di
                               // scrittura.
                                pwrite=0;
                }
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione InitScan si occupa di inizializzare le variabili //
//  per la scansione del segnale e la compilazione degli header.  //
//  La funzione ritorna la quantità di informazione processata.   //
//                                                                //
////////////////////////////////////////////////////////////////////
void Aceas::InitScan(unsigned char *buffer)
{
        int i;

        #ifndef _ACE_EVOLUTION
    // Richiede il puntatore all'headertime
        header_time=GETPOINTER(0,ID_HEADERTIME,Headers,list);
        #endif // _ACE_EVOLUTION
        for (i=0;i<nlist
            #ifdef _ACE_EVOLUTION
            +1
            #endif
        ;i++)
            #ifndef _ACE_EVOLUTION
                                        header_value[i]=(unsigned    short*)
GETPOINTER(i,ID_HEADERVALUE,Headers,list);
            #else
            header_value[i]=&evolution_buffer[i][0];
            #endif

        // Copia la misura corrente per un'utilizzazione futura
        // nella funzione Scan.
        memcpy(previous,buffer,bytemisura);
        // Copia il segnale temporale iniziale nell'headertime.
        INITFIRSTHEADERTIME(buffer,(unsigned short *) header_time);
        // Copia il segnale temporale binario ritornato dalla
        // funzione precedente per un'utilizzazione futura
        // nella funzione Scan.
        memcpy(binary1,header_time,HEADER_TIME_SIZE);
        // Copia nell'headermeasure di ogni segnale il valore iniziale.
```

```
        for   (i=0;i<nlist;i++)   CONVERTMEASURETONUMBER(buffer+offset[i],(unsigned
short*)
                                                #ifndef _ACE_EVOLUTION
                                                GETPOINTER(i,ID_MEASURE,Headers,list)
                                                #else
                                                &header_measures[i][0]
                                                #endif //_ACE_EVOLUTION
                                                ,list[i]);
        // Indica che una misura è stata processata.
        header_aceas.nmisure=1;
        // Indica che l'operazione di inizializzazione è stata compiuta.
        initscan=TRUE;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione Scan si occupa della compilazione degli header.  //
//                                                                //
////////////////////////////////////////////////////////////////////
BOOL Aceas::Scan(unsigned char *buffer,unsigned short size)
{
        unsigned short i=0;
        unsigned short j;

        // Ritorna TRUE se il buffer è vuoto
        if (!size) return TRUE;
        // Inizializza gli header e le variabili processando solo
        // la prima misura valida.

        // Aggiorna il checksum delle misure
        header_aceas.crc += MISCRC(buffer,size);

        if (!TESTMISURA(buffer,String,bytemisura,size)) return FALSE;

        unsigned short r=write(tmp,buffer,size);

        if (r != size) return FALSE;

        if (initscan == FALSE) {
                                InitScan(buffer);
                                i += bytemisura;
        }
        // Processa le misure disponibili nel buffer di ingresso.
        while (i < size  && size-i >= bytemisura) {
                        // Incrementa il numero di misure valide processate.
                        header_aceas.nmisure++;
                        // Aggiorna l'headertime processando il segnale
                        // temporale della misura selezionata.
                        #ifdef _ACE_EVOLUTION
                        if (nevolution_buffer[nlist] >= MAX_LINES) return FALSE;
                        #else
                        if (header_time[12] >= NDELTA) return FALSE;
                        #endif

                              FORMATHEADERTIME(binary1,buffer+i
                    #ifdef _ACE_EVOLUTION
                    ,header_value[nlist],nevolution_buffer[nlist]
                    #else
                    ,header_time
                    #endif
                    );
                              // Processa ogni singolo segnale della misura
                              for (j=0;j<nlist;j++) {
                                      // Identifica il tipo di codifica richiesta
                                      // per il segnale selezionato.
                                      if (type[j] == IDML_DELAY) {
                                      // calcola la derivata del segnale.

CALCDELTAMEASURE(previous+offset[j],buffer+i+offset[j],delta_measure,list[j],nodigit
+n_nodigit-1,previous);
                                                                }
                                                                else
                                                                {
                                      // Copia la misura totalmente.
                                      delta_measure[0]=0;
```

```
memcpy(delta_measure+1,buffer+i+offset[j],list[j]);
                                                                }
                                    // Aggiorna l'headervalue del segnale selezionato
                                    // attraverso le informazioni memorizzate nel
                                    // buffer delta_measure
                                    #ifdef _ACE_EVOLUTION

FORMATHEADER(header_value[j],nevolution_buffer[j],delta_measure,list[j]+1);
                                    #else

FORMATHEADER(header_value[j],delta_measure,list[j]+1);
                                    #endif // _ACE_EVOLUTION
                                                }
                                    // Processa il segnale successivo.
                  // Copia la misura binaria corrente per utilizzarla
                  // in coppia con la prossima per il calcolo della
                  // derivata. L'inizializzazione del buffer 'previous'
                  // avviene in InitScan.
                  memcpy(previous,buffer+i,bytemisura);
                  // Selezione la misura successiva
                  i += bytemisura;
                           } // while
          return TRUE;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione InitComp inizializza le variabili per la          //
//  compressione del segnale.                                        //
//  La funzione ritorna la quantità di informazione processata    //
//                                                                //
////////////////////////////////////////////////////////////////////
bool Aceas::InitComp(unsigned char *buffer)
{
          int i;

          #ifndef _ACE_EVOLUTION
          unsigned short t;

          // Richiede il puntatore all'headertime.
          header_time=GETPOINTER(0,ID_HEADERTIME,Headers,list);
          // Richiede il puntatore alla tabella del segnale temporale
          table_time=GETPOINTER(0,ID_TABELLA,Headers,list);
          #endif // _ACE_EVOLUTION
          // Se ci sono più misure bisogna compilare le tabelle
          // per utilizzare le funzioni di compressione.
          if (header_aceas.nmisure > 1) {
     #ifndef _ACE_EVOLUTION
          // Compila la tabella temporale.
          COMPTABTIME(header_time,table_time,list[0]+1);
          #endif
     // Compila la tabella per ciascun segnale.
          for (i=0;i<nlist
                  #ifdef _ACE_EVOLUTION
                  +1
                  #endif
          ;i++) {
                              #ifndef _ACE_EVOLUTION
                              // Richiede il puntatore all'headervalue da convertire
                              // in tabella.
                              header_value[i]=(unsigned                          short*)
GETPOINTER(i,ID_HEADERVALUE,Headers,list);
                              // Richiede il puntatore alla tabella che si vuole
               // compilare.
                              table_value[i]=GETPOINTER(i,ID_TABELLA,Headers,list);
               // Compila la tabella del segnale.
                              COMPTAB(header_value[i],table_value[i],list[i]+1);
                              // Il segnale temporale viene processato
                              // insieme al primo segnale disponibile.
                              // Il numero di celle della prima tabella
                              // sarà incrementato di 1.
                              if (!i) t=1; else t=0;
                              // Calcola la lunghezza della codeword di
               // memorizzazione per ciascun segnale.
```

```
                        lbit[i]=LENG(table_value[i],list[i]+t+1);
                        #else
                        header_value[i]=&evolution_buffer[i][0];
                        if      (!quickSort(header_value[i],nevolution_buffer[i]))
return false;
                        #endif // _ACE_EVOLUTION
                              }
        // Inizializza le variabile utilizzate dalla funzione STORE
    SW=0;
        BIT_S=0;

        // Copia il segnale temporale iniziale per essere utilizzato
        // in Compress.
        memcpy(binary1,header_time,HEADER_TIME_SIZE);
                                    } // Questa parentesi chiude l'istruzione
                                      // if iniziale.
        // Scrive l'intestazione del file.
        header_aceas.leng_ml=Strlen(String)+1;
        write(drain,(unsigned char *) &header_aceas,sizeof(header_aceas));
        // Scrive la stringa di definizione nel file.

        write(drain,String,header_aceas.leng_ml);
        // Scrive gli header utilizzati nel file.
        SaveHeaders(drain);
        #ifdef _ACE_EVOLUTION
        for (i=0;i<nlist+1;i++) {
       huffman(header_value[i],nevolution_buffer[i]);
       huffmanFill(header_value[i],0,0,0);
       huffmanCompact(header_value[i],nevolution_buffer[i]);
        }
        #endif
        // Calcola il checksum per la prima misura valida.
        // Indica che una misura è stata processata.
        // Se non ci sono misure il programma non arriva a questo punto.
        header_aceas.nmisure=1;

        // Converte la misura da ASCII in binario e la conserve per
       // essere utilizzata in Compress.
        memcpy(previous,buffer,bytemisura);
        // Indica che l'inizializzazione ha avuto esito positivo
        initcomp=TRUE;
       // Indica il numero di misure processate dal buffer.
    return true;
}




////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione Compress esegue la compressione del segnale       //
//                                                                //
////////////////////////////////////////////////////////////////////
unsigned short Aceas::Compress(unsigned char *buffer,unsigned short size)
{
        unsigned short i=0,j;
        unsigned short v;
        #ifdef _ACE_EVOLUTION
        int index;
        #else
        unsigned short t;
        #endif


        // Ritorna 0 se il file di uscita non è stato creato.
        if (!drain) return 0;
        // Inizializza le variabile per la compressione, compila le
        // tabelle e scrive gli header nel file di uscita.

        if (initcomp == FALSE) {
               lseek(tmp,0,SEEK_SET);
                              size=read(tmp,buffer,size);
                              // Ritorna se il buffer di ingresso è vuoto.
                              if (!size || size == 0xffff) return 0;
```

```
                                if (!InitComp(buffer)) return 0;
                                i += bytemisura;
                              }
                              else
                                size=read(tmp,buffer,size);
        // Ritorna 0 se il buffer di ingresso è vuoto.
        if (!size) return 0;
        // Processa ogni misura.

        while (i < size && size-i >= bytemisura) {
                        // Incrementa il numero di misura processate.
                        header_aceas.nmisure++;

                        #ifdef _ACE_EVOLUTION
                        // Calcola la variazione temporale e la converte in un
indice compreso tra zero e NDELTA.
                        j=nlist;

index=GETNDELTA(binary1,buffer+i,&evolution_buffer[j][0],nevolution_buffer[j]);
            *((unsigned long*) temp_buffer)=(header_value[j])[index].hd;
                        lbit[j]=(header_value[j])[index].lbit;
                        if (lbit[j]) {
                v=STORE((unsigned short*) wr,(unsigned short*) temp_buffer,lbit[j]);
                // Memorizza nel buffer temporaneo i bit
                // significativi.
                WriteToBuffer(drain,(unsigned char*) wr,v,TRUE);
                        }
                        #else
                        // Calcola la varizione temporale e la converte in un indice
compreso tra zero e NDELTA.
                        delta=GETNDELTA(binary1,buffer+i,header_time);
                        // Ingloba l'indice ricavato nella stringa del segnale.
                        delta_measure[list[0]+1]=delta;
                        #endif

                        // Processa ogni singolo segnale.
                        for (j=0;j<nlist;j++) {
                #ifndef _ACE_EVOLUTION
                                // La lunghezza della stringa binaria del primo
segnale è aumentata di 1 per la presenza del segnale temporale.
                                if (!j) t=1; else t=0;
                #endif
                                // Decide il tipo di codifica da effettuare.
                                if (type[j] == IDML_DELAY) {
                                                //  Calcola  la  derivata  del
segnale.

CALCDELTAMEASURE(previous+offset[j],buffer+i+offset[j],delta_measure,list[j],nodigit
+n_nodigit-1,previous);
                                }
                                                                else
                                                                {
                                                //      Copia      il      segnale
integralmente.
                                                delta_measure[0]=0;

memcpy(delta_measure+1,buffer+i+offset[j],list[j]);
                                                                }
                                // Comprime la stringa binaria generando una
codeword
                                // di lunghezza lbit[j].
                                #ifndef _ACE_EVOLUTION
                                COMPRESS(table_value[j],(unsigned short*)
temp_buffer,(unsigned char *) delta_measure,list[j]+t+1);
                                #else

index=COMPRESS(header_value[j],nevolution_buffer[j],(unsigned char *)
delta_measure,list[j]+1);
                                *((unsigned long*)
temp_buffer)=(header_value[j])[index].hd;
                                lbit[j]=(header_value[j])[index].lbit;
                                #endif // _ACE_EVOLUTION
                                // Comprime i bit significativi eliminando
                                // quelli in eccesso.
                                if (!lbit[j]) continue;
```

```
                                        v=STORE((unsigned  short*)  wr,(unsigned  short*)
temp_buffer,lbit[j]);
                                        // Memorizza nel buffer temporaneo i bit
                                        // significativi.
                                        WriteToBuffer(drain,(unsigned char*) wr,v,TRUE);
                                                }
                                // Copia la misura corrente per essere utilizzata
                                // in coppia con la successiva. L'inizializzazione
                                // del buffer 'previous' avviene in InitComp.
                        memcpy(previous,buffer+i,bytemisura);
                        // Seleziona la misura successiva.
                        i += bytemisura;
                                }
                return size;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione CloseComp chiude il file e la compressione.       //
//                                                                //
////////////////////////////////////////////////////////////////////
void Aceas::CloseComp(void)
{
                unsigned short v;

                // Ritorna se non ci sono file di uscita aperti.
                if (!drain) return;
                // Memorizza nel buffer temporaneo ciò che è
                // rimasto nelle varibili usate da STORE.
                v=STORE((unsigned short*) wr,0,0);
                // Libera il buffer temporaneo.
                WriteToBuffer(drain,(unsigned char*) wr,v,TRUE,TRUE);
                #ifdef ENABLE_ACEAS_DEBUG
        printf("File size: %ld\n",lseek(drain,0L,SEEK_CUR));
        printf("Measures %ld\n",header_aceas.nmisure);
        #endif
                // Si posiziona all'inizio del file
                lseek(drain,0,SEEK_SET);
                // Aggiorna l'intestazione del file compresso.
                write(drain,(unsigned char *) &header_aceas,sizeof(header_aceas));
                // Chiude il file di uscita.
                close(drain);
                drain=0;
                close(tmp);
                tmp=0;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione Compress si occupa di comprimere un file     //
//  contenente il segnale. La funzione si appoggia alla stringa //
//  di definizione 'string'.                                        //
//  La funzione ritorna un FALSE se la compressione non è         //
//  riuscita.                                                     //
//                                                                //
////////////////////////////////////////////////////////////////////
BOOL Aceas::Compress(char *sourcefile,char *drainfile,unsigned char *string)
{
                unsigned short q;
                unsigned short swapsize;
        char model[256];

                // Apre il file di ingresso
                source=open(sourcefile,O_BINARY | O_RDONLY);
                // Ritorna FALSE se il file non è stato trovato
                if (source == -1) return(FALSE);

                if (!string) {
                                read(source,model,sizeof(model));
                                parser(String,model);
                                lseek(source,0,SEEK_SET);
                                string=String;
                        }

                // Alloca la memoria per l'elaborazione del segnale.
```

```
        if (AllocHeaders(string) == FALSE) { close(source); return FALSE; }

        // Calcola la dimensione effettiva del buffer temporaneo
        // in funzione della lunghezza della misura da processare.
        swapsize=(SWAPSIZE/bytemisura);
        swapsize *= bytemisura;

        // Inizializza le variabili e apre il file di uscita.
        OpenComp((unsigned char *) drainfile);

        // Esegue la scansione del file di ingresso e
        // compila gli header.
        while(TRUE) {
                q=read(source,bufferin,swapsize);
                if (!q) break;
                if (!Scan(bufferin,q)) return FALSE;
                    }

        // Esegue la compressione del file di ingresso
        while(Compress(bufferin,swapsize));

        // Chiude il file di uscita
        CloseComp();
        // Libera la memoria allocata per la compressione
        FreeHeaders();
        // Chiude il file di ingresso.
        close(source);
        source=0;
        return(TRUE);
}

//////////////////////////////////////////////////////////////////
//                                                              //
//  La funzione Decompress si occupa di decomprimere un file    //
//  precedentemente compresso.                                    //
//  LA funzione ritorna TRUE se la decompressione ha avuto esito //
//  positivo, FALSE se ci sono errori, NOMISURE se non ci sono  //
//  misure sul file.                                            //
//                                                              //
//////////////////////////////////////////////////////////////////
BOOL Aceas::Decompress(char *sourcefile,char *drainfile)
{
        int i,n=0;
        BOOL end_file=FALSE;
        unsigned short j;
        HEADER_ACEAS header;
        int q=0;

        #ifdef _ACE_EVOLUTION
    memset(evolution_buffer,0,sizeof(evolution_buffer));
    memset(nevolution_buffer,0,sizeof(nevolution_buffer));
    memset(header_value,0,sizeof(header_value));
    memset(header_time,0,sizeof(header_time));
    memset(header_measures,0,sizeof(header_measures));
    #else
    unsigned short t;
    #endif

    // Inizializza le variabili usate dalla funzione GET
        GR=0;
        BIT_R=0;

    // Azzera il puntatore del buffer temporaneo.
        pwrite=0;

    // Azzera il checksum ed il numero di misure estratte
        header_aceas.crc=0;
        header_aceas.nmisure=0;
    // Apre il file di ingresso.
        source=open(sourcefile,O_BINARY | O_RDONLY);
        if (source == -1) return FALSE;
    // Legge l'intestazione del file aperto.
        q=read(source,(unsigned char *) &header,sizeof(header));
        if (q < (int)sizeof(header)) {
            return FALSE;
```

```
        }
        // Confronta la segnatura per verificare che si tratti
        // di un file compresso con la release 4 del algoritmo
        // ACE4.
        if (strcmp(header.signature,header_aceas.signature) != 0) {
                                                                close(source);
                                                                source=0;

return(NOACEFILE);
                                                                 }

    // Legge il numero di misure dal file.
        if (!header.nmisure) return NOMEASURE;
        // Legge la stringa di definizione dal file compresso.
        read(source,String,header.leng_ml);

        // Alloca la memoria necessaria per l'elaborazione
        // del file.
        FreeHeaders();
        if (AllocHeaders(String) == FALSE) {
                                        close(source);
                                        source=0;
                                        return(FALSE);
                                          }
    #ifdef _ACE_EVOLUTION
    for (i=0;i<nlist+1;i++) header_value[i]=&evolution_buffer[i][0];
    #endif
    // Legge header e valori iniziali dal file compresso.
        LoadHeaders(source,header_time);

        #ifdef __WINDOWS__
        writeLog((unsigned char*) "ACE: LoadHeader");
        #endif

       // Crea il file di uscita.
    #ifndef _LINUX
        _fmode=O_BINARY;
        #endif
        drain=open(drainfile, O_CREAT|O_WRONLY|O_TRUNC
            #ifdef _LINUX
            ,S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
            #endif
            );

        #ifdef __WINDOWS__
        char s[64];

        sprintf(s,"ACE: %d %d",drain,errno);
        writeLog((unsigned char*) s);
    #endif
    #ifndef _ACE_EVOLUTION
        // Acquisisce i puntatori alla tabella temporale e
    // all'header temporale.
    unsigned char  *header_time;
    unsigned char *table_time;
        table_time=GETPOINTER(0,ID_TABELLA,Headers,list);
        header_time=GETPOINTER(0,ID_HEADERTIME,Headers,list);

        // Compila la tabella temporale se ci sono
    // più di una misura.
        if (header.nmisure > 1) DECTABTIME(header_time,table_time,list[0]+1);
        #endif // _ACE_EVOLUTION

        // Memorizza il valore temporale iniziale per un
        // utilizzo futuro.
    memcpy(binary1,header_time,HEADER_TIME_SIZE);

    #ifndef _ACE_EVOLUTION
        for (i=0;i<nlist;i++) {
                table_value[i]=GETPOINTER(i,ID_TABELLA,Headers,list);
                header_value[i]=(unsigned                                   short*)
GETPOINTER(i,ID_HEADERVALUE,Headers,list);
        }

    // Compila la tabella per ogni singolo segnale
```

```
        for (i=0;i<nlist;i++) {
                DECTAB(header_value[i],table_value[i],list[i]+1);

                // La lunghezza della prima tabella è incrementata
                // di 1 in quanto memorizza anche il segnale
        // temporale.
                if (!i) t=1; else t=0;
                // Calcola la lunghezza della codeword per ogni
        // singolo segnale.
                lbit[i]=LENG(table_value[i],list[i]+t+1);
    }
    #endif

        // Estrae i valori iniziali del segnale temporale e di
        // ogni altro segnale dagli header.
        for (j=0;j<nlist;j++) {
                if (!j) INITFIRSTTEXTTIME((unsigned char*) wr,(unsigned short*)
header_time);
                CONVERTNUMBERTOMEASURE((unsigned short*)
                        #ifdef _ACE_EVOLUTION
                        &header_measures[j][0]
                        #else
                        GETPOINTER(j,ID_MEASURE,Headers,list)
                        #endif // _ACE_EVOLUTION
                        ,(unsigned char*) wr+offset[j],list[j]);
                              }

        // Conserva la misura binaria per un utilizzo futuro.
        memcpy(previous,wr,bytemisura);
        // Aggiorna il checksum locale.
        // Incrementa il numero di misure estratte.
        header_aceas.nmisure++;
        // Scrive la misura del buffer temporaneo.
        WriteToBuffer(drain,(unsigned char*) wr,bytemisura,FALSE);
        // Il valore di 'i' indica la quantità di buffer libero
        // in cui è possibile caricare informazione dal file.
        i=SWAPSIZE;
        // Se il numero di misure è minore di due la decompressione
        // si arresta.
        if (header.nmisure > 1) {
do {
        // Il valore di n indica in questa posizione i numero di byte
        // presenti nel buffer di lettura.
        if (n < bytemisura && !end_file) {
                                        // Legge dal file tanti byte
                                        // quanti indicati dalla variabile
                                        // 'i'
                                        memcpy(bufferin,bufferin+q,n);
                    q=0;
                                        i=read(source,bufferin+n,i);
                                                }
                                                 else
                                                 // indica che nessun byte
                                               // può essere letto dal file
                                                 { i=-1; }
        // Da questa posizione i indica il numero di caratteri
        // letti dal file.
        // Se i == 0 la fine del file è stata raggiunta
        if (!i) end_file=TRUE;
        if (i == -1) {
                // Nessun carattere è stato letto.
                        i = n;
                }
                else
                {
                    // I caratteri letti vengono aggiunti a quelli
                  // presenti nel buffer.
                    i += n;
                }
        // Il valore di 'i' indica da questo punto il numero di byte
        // presenti nel buffer di lettura.

    #ifdef _ACE_EVOLUTION
    if (nevolution_buffer[nlist] > 1) {
              n=GET(header_value[nlist],delta_measure,list[nlist],(unsigned short*)
```

```
(bufferin+q),i);
            if (n > i) {
                        close(source);
                        close(drain);
                        source=0;
                        drain=0;
                        return(FALSE);
                       }
            q += i-n;
            if (n < bytemisura && !end_file) {
                                        memcpy(bufferin,bufferin+q,n);
                    q=0;
                                        i=read(source,bufferin+n,SWAPSIZE-n);
                                        if (!i) end_file=TRUE;
                                        i += n;
                                        n = i;
                                    }
            i=n;
        } else memcpy(delta_measure,evolution_buffer[nlist][0].v,list[nlist]);
    GETTIME(binary1,(unsigned char*) wr,(unsigned short *) delta_measure);

    #endif

         for (j=0;j<nlist;j++) {
        #ifdef _ACE_EVOLUTION
        if (nevolution_buffer[j] > 1) {
                    n=GET(header_value[j],delta_measure,list[j]+1,(unsigned short*)
(bufferin+q),i);
            if (n > i) {
                        close(source);
                        close(drain);
                        source=0;
                        drain=0;
                        return(FALSE);
                       }
            q += i-n;
        } else memcpy(delta_measure,evolution_buffer[j][0].v,list[j]+1);

        #else
        if (!j) t=1; else t=0;
                 // Viene estratta la codeword dalla serie di bit
                 // letta dal file. Ogni segnale possiede una codeword differente.
                 if (lbit[j]) {
                              n=GET((unsigned short*) (bufferin+q),(unsigned short*)
temp_buffer,lbit[j],i);
                // La funzione GET in caso di errore ritorna un valore
                // di byte presenti nel buffer superiore a quello fornito.
                if (n > i) {
                        close(source);
                        close(drain);
                        source=0;
                        drain=0;
                        return(FALSE);
                       }
                q += i-n;
                 }
                 // Richiede il puntatore alla tabella del segnale corrente.
                 // La tabella del primo segnale ingloba anche la tabella
        // temporale.
                 if (!j) t=1; else t=0;
        // Trasforma la codeword in stringa binaria.
                 DECOMPRESS(table_value[j],(unsigned                                short*)
temp_buffer,delta_measure,list[j]+t+1);
                 #endif
        // Identifica il contenuto del buffer delta_measure.
                 if (type[j] == IDML_DELAY) {
                                    // Calcola la misura dalla derivata.

CALCMEASURE(previous+offset[j],delta_measure,(unsigned                             char*)
wr+offset[j],list[j],nodigit+n_nodigit-1,previous);
                                              }
                                            else
                                              {
                                    // Copia la misura integralmente.
                                    memcpy(wr+offset[j],delta_measure+1,list[j]);
```

```
                                        }
        #ifndef _ACE_EVOLUTION
        // Calcola il segnale temporale della misura.
                if          (!j)          GETTIME(binary1,(unsigned          char*)
wr,header_time,delta_measure[list[0]+1]);
                #endif
                // Legge altri byte dal file in ingresso per rifornire
                // il buffer di elaborazione. Il numero di byte letti
               // è puramente arbitrario.
                if (n < bytemisura && !end_file) {
                                        memcpy(bufferin,bufferin+q,n);
                   q=0;
                                        i=read(source,bufferin+n,SWAPSIZE-n);
                                        if (!i) end_file=TRUE;
                                        i += n;
                                        n = i;
                                    }
                i=n;
                                }
         // Calcola il numero di byte necessari per completare il buffer
         // di ingresso.
         i=SWAPSIZE-n;
         // Memorizza la misura corrente per essere usata in coppia con la
         // successiva. L'inizializzazione del buffer avviene all'inizio
         // della funzione.
         memcpy(previous,wr,bytemisura);
         // Calcola il checksum della misura.
         // Incrementa il numero delle misure estratte.
         header_aceas.nmisure++;
         // Scrive la misura nel buffer temporaneo.
         WriteToBuffer(drain,(unsigned char*) wr,bytemisura,FALSE);
         // Rivela la fine della decompressione.
         if  (i  ==  SWAPSIZE  &&  end_file  ==  TRUE  &&  header_aceas.nmisure  ==
header.nmisure) break;
   } while(1);
                                }
         // Pulisci il buffer di uscita
         WriteToBuffer(drain,(unsigned char*) wr,0,FALSE,TRUE);
         // Chiude i file di ingresso e di uscita.
         close(source);
         close(drain);
         // Libera la memoria allocata.
         FreeHeaders();
         source=0;
         drain=0;
         // Ritorna un errore se il checksum locale è differente da quello
         // memorizzato nel file compresso.
         if (header_aceas.crc != header.crc) return(FALSE);

         return(TRUE);
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione GetList si occupa di analizzare la stringa        //
//  di definizione estraendo le informazioni relative ai          //
//  segnali presenti nella stringa di misura.                         //
//  La funzione ritorna il numero di segnali presenti nella       //
//  stringa di misura.                                            //
//                                                                //
////////////////////////////////////////////////////////////////////
short  int  GetList(unsigned  char  *string,char  *list,char  *offset,unsigned  char
*type,unsigned short &size,unsigned char *nodigit,unsigned short &n_nodigit)
{
         int i=0;
         int j=0,k;
         unsigned char min;

         // Inizializza il buffer delle cifre interne a stringhe numeriche
         // che si comportano da costanti.
         n_nodigit=0;

         do {
                 switch(string[i]) {
                                 case IDML_TIME:
```

```
                                        i++;
                                        YEAR_OFFSET=(unsigned short) string[i];
                                        i++;
                                        YEAR_LENGTH=(unsigned short) string[i];
                                        i++;
                                        MONTH_OFFSET=(unsigned short) string[i];
                                        i++;
                                        DAY_OFFSET=(unsigned short) string[i];
                                        i++;
                                        HOUR_OFFSET=(unsigned short) string[i];
                                        i++;
                                        MIN_OFFSET=(unsigned short) string[i];
                                        i++;
                                        SEC_OFFSET=(unsigned short) string[i];
                                        break;
                                case IDML_VALUE:
                                        i++;
                                        type[j]=string[i];   // Tipo di codifica.
                                        i++;
                                        offset[j]=string[i];   //   Offset   della
stringa
                                        i++;                    // numerica.
                                        list[j]=string[i];    // Lunghezza della
stringa
                                        j++;                    // numerica.
                                        break;
                                 case IDML_SUB:                // Non hanno alcun
valore
                                case IDML_COST:              // e vengono saltate.
                                        i += 2;
                                        break;
                                case IDML_RANGE:             // Viene saltata.
                                        i += 3;
                                        break;
                                 case IDML_SIZE:                 // Acquisisce la
lunghezza
                                        i++;                     // della stringa di
misura
                                        size=string[i];
                                        break;
                                case IDML_NODIGIT:
                                        i++;
                                        nodigit[n_nodigit]=string[i];  //  Offset
del carattere
                                        n_nodigit++;                              //
IDML_NODIGIT.
                                        break;
                                }
                        i++;
                } while(string[i] != IDML_END);  // se string[i] == IDML_END viene trovata
                                                 // una IDML_END e l'elaborazione
                                                 // si arresta.

        // Ordina gli offset dei IDML_NODIGIT dal più piccolo al più grande.
        for (k=0;k<n_nodigit;k++) {
        for (i=k;i<n_nodigit;i++) {
                                if (nodigit[i] < nodigit[k]) {
                                                min=nodigit[i];
                                                nodigit[i]=nodigit[k];
                                                nodigit[k]=min;
                                                      }
                                }
                                }
        for (k=n_nodigit;k>0;k--) nodigit[k]=nodigit[k-1];
        nodigit[0]=255;
        n_nodigit++;
        #ifdef _ACE_EVOLUTION
        list[j]=12;
        #endif

        return j;
}

//////////////////////////////////////////////////////////////////
//                                                              //
```

```
//  La funzione AllocHeaders si occupa di allocare la memoria    //
//  necessaria alla compressione.                                //
//  La funzione ritorna FALSE se ci sono stati errori.           //
//                                                               //
///////////////////////////////////////////////////////////////////
BOOL Aceas::AllocHeaders(unsigned char *string)
{
        // Copia la stringa di definizione nel buffer della classe
        // per usi futuri.
        Strcpy(String,string);
        #ifndef _ACE_EVOLUTION
        // Calcola lo spazio necessario per l'headertime e per la
    // tabella temporale.
        unsigned short size=(HEADER_TIME_SIZE+1+(NDELTA*HEADER_TIME_SIZE))+NDELTA+1;
        unsigned short i;
        #endif
        // Ritorna FALSE se la memoria è stata già allocata.
        if (temp_buffer) return(FALSE);
        // Processa la stringa di definizione riconoscendo
        // i segnali presenti nella stringa di misura
        // e gli eventuali IDML_NODIGIT.
        nlist=GetList(string,(char*)                                    list,(char*)
offset,type,bytemisura,nodigit,n_nodigit);
        // Se non ci sono segnali da processare ritorna FALSE
        if (!nlist) return FALSE;
        #ifndef _ACE_EVOLUTION
        // Calcola lo spazio necessario per header e tabelle
        for (i=0;i<nlist;i++) {
                        // spazio per le tabelle e per i header
                                size += ((list[i]+1)*13);
                              // spazio per il valore iniziale
                                size += (NWORD*2);
                             }
        // Alloca la memoria richiesta.
        Headers=new unsigned char[size];
        // Se ci sono errori ritorna FALSE.
        if (!Headers) return(FALSE);
        #endif
        // Alloca lo spazio per i buffer temporanei
        temp_buffer=new char[bytemisura];
        if (!temp_buffer) {
                #ifndef _ACE_EVOLUTION
                                delete[] (Headers);
                #endif
                temp_buffer=0;
                                return(FALSE);
                             }
        wr=new char[bytemisura];
        if (!wr) {
            #ifndef _ACE_EVOLUTION
                        delete[] (Headers);
            #endif
                        delete[] (temp_buffer);
                        temp_buffer=0;
            return(FALSE);
                      }
       // Inizializza la memoria ricevuta.
    #ifndef _ACE_EVOLUTION
        memset(Headers,0,size);
        #endif
        memset(temp_buffer,0,bytemisura);
        memset(wr,0,bytemisura);
        // Memorizza la lunghezza della stringa di definizione
        header_aceas.leng_ml=Strlen(string)+1;
        #ifndef _ACE_EVOLUTION
        SHeaders=size;
        #endif

        return(TRUE);
}



///////////////////////////////////////////////////////////////////
```

```
//                                                                    //
//  La funzione FreeHeaders si occupa di liberare la memoria          //
//  precedentemente allocata da AllocHeaders.                            //
//                                                                    //
////////////////////////////////////////////////////////////////////////
void Aceas::FreeHeaders(void)
{
    // Ritorna se non è stata allocata memoria
        if (!temp_buffer) return;
    #ifndef _ACE_EVOLUTION
        // Libera la memoria allocata
        delete[](Headers);
        #endif
        delete[] temp_buffer;
        delete[] wr;
        temp_buffer=0;
}

////////////////////////////////////////////////////////////////////////
//                                                                    //
//  La funzione SaveHeaders si occupa di scrivere gli header          //
//  necessari alla compressione nel file di uscita.                      //
//                                                                    //
////////////////////////////////////////////////////////////////////////
void Aceas::SaveHeaders(unsigned short handle)
{
        unsigned short j;

        #ifdef _ACE_EVOLUTION

        unsigned short i;
        // Memorizza nel file l'headermeasure del primo segnale
        write(handle,header_measures,2*NWORD);
        write(handle,header_time,HEADER_TIME_SIZE);
        write(handle,nevolution_buffer,(nlist+1)*sizeof(unsigned short));
        for (i=0;i<nlist+1;i++) {
            #ifdef ENABLE_ACEAS_DEBUG
            printf("Deltas %d: %d\n",i,nevolution_buffer[i]);
            #endif
            for (j=0;j<nevolution_buffer[i];j++) {
                        if (i == nlist) CONVERTTIMETONUMBER((unsigned short*)
evolution_buffer[i][j].v,(unsigned long*)evolution_buffer[i][j].bin);
                                else
                                        CONVERTMEASURETONUMBER((unsigned char*)
evolution_buffer[i][j].v,evolution_buffer[i][j].bin,list[i]+1);

write(handle,evolution_buffer[i][j].bin,sizeof(evolution_buffer[i][j].bin));

write(handle,&evolution_buffer[i][j].n,sizeof(evolution_buffer[i][j].n));
            }
                        // Memorizza nel file l'headermeasure del segnale
            // selezionato.
                        write(handle,&header_measures[i][0],2*NWORD);
                        // Si sposta all'inizio dell'headervalue successivo.
                        //n += 2*NWORD;
                          }
        #else
        // Indica l'offset dell'headervalue del secondo segnale
        unsigned                                                        short
n=HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+((HEADER_TIME_SIZE+1)*(list[0]+1))+ND
ELTA+1+2*NWORD;
    // Memorizza nel file l'headertime e l'headervalue del primo segnale.

write(handle,Headers,HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+(2*(list[0]+1)));
        // Memorizza nel file l'headermeasure del primo segnale

write(handle,Headers+HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+((HEADER_TIME_SIZE
+1)*(list[0]+1))+NDELTA+1,2*NWORD);
        for (j=1;j<nlist;j++) {
                        // Memorizza nel file l'headervalue del segnale
                       // selezionato.
                        write(handle,Headers+n,(list[j]+1)*2);
                        // Sposta il puntatore oltre la tabella.
                        n += ((list[j]+1)*13);
                        // Memorizza nel file l'haedermeasure del segnale
```

```
                        // selezionato.
                         write(handle,Headers+n,2*NWORD);
                         // Si sposta all'inizio dell'headervalue successivo.
                         n += 2*NWORD;
                           }
    #endif

    #ifdef ENABLE_ACEAS_DEBUG
    printf("Header size: %ld\n",lseek(drain,0L,SEEK_CUR));

    #ifdef _ACE_EVOLUTION
    int hour=0,minute=0,second=0,day=1;

        FILE *drain=fopen("header.txt","wt+");
        for (i=0;i<nevolution_buffer[0];i++) {
                            fprintf(drain,"01-%2.2hd-2019   %2.2hd:%2.2hd:%2.2hd
",day,hour,minute,second);

                                                        for     (j=0;j<list[0];j++)
fprintf(drain,"%c",evolution_buffer[0][i].v[j]+'0');
            fprintf(drain," %5.5d\r\n",evolution_buffer[0][i].n);
            second++;
                        if (second == 60) {
                                minute++;
                                second=0;
                                if (minute == 60) {
                                        minute=0;
                                        hour++;
                                        if (hour == 24) {
                                                day++;
                                                hour=0;
                                        }
                                }
                        }
        }
        fclose(drain);
        #endif
    #endif
}

//////////////////////////////////////////////////////////////////////
//                                                                  //
//  La funzione LoadHeaders si occupa di recupare header            //
//  necessari alla decompressione dal file di uscita.                  //
//                                                                  //
//////////////////////////////////////////////////////////////////////
void Aceas::LoadHeaders(unsigned short handle
                        #ifdef _ACE_EVOLUTION
                        ,unsigned short *header_time
                        #endif
                        )
{
        unsigned short j;

    #ifdef _ACE_EVOLUTION
    unsigned short i,tmp;
        // Legge dal file l'headermeasure del primo segnale.
        read(handle,header_measures,2*NWORD);
        read(handle,header_time,HEADER_TIME_SIZE);
        read(handle,nevolution_buffer,(nlist+1)*sizeof(unsigned short));
        for (i=0;i<nlist+1;i++) {
            for (j=0;j<nevolution_buffer[i];j++) {

read(handle,evolution_buffer[i][j].bin,sizeof(evolution_buffer[i][j].bin));

read(handle,&evolution_buffer[i][j].n,sizeof(evolution_buffer[i][j].n));
                            if (i == nlist) CONVERTTIMETOBINARY((unsigned short*)
evolution_buffer[i][j].v,(unsigned long*) evolution_buffer[i][j].bin);
                                    else

CONVERTNUMBERTOMEASURE(evolution_buffer[i][j].bin,(unsigned                 char*)
evolution_buffer[i][j].v,list[i]+1);
            }
            tmp=nevolution_buffer[i];
            huffman(&evolution_buffer[i][0],nevolution_buffer[i]);
```

```
            nevolution_buffer[i]=tmp;
            huffmanFill(&evolution_buffer[i][0],0,0,0);

                        // Legge dal file l'headermeasure del segnale
                        // selezionato.
                        read(handle,&header_measures[i][0],2*NWORD);
                          }
    #else
    // Indica l'offset dell'headervalue del secondo segnale
        unsigned                                                    short
n=HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+((HEADER_TIME_SIZE+1)*(list[0]+1))+ND
ELTA+1+2*NWORD;

    // Legge dal file l'headertime e l'headervalue del primo segnale.

read(handle,Headers,HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+(2*(list[0]+1)));
        // Legge dal file l'headermeasure del primo segnale.

read(handle,Headers+HEADER_TIME_SIZE+1+(HEADER_TIME_SIZE*NDELTA)+(13*(list[0]+1))+ND
ELTA+1,2*NWORD);
        for (j=1;j<nlist;j++) {
                        // Legge dal file l'headervalue del segnale
                       // selezionato.
                        read(handle,Headers+n,(list[j]+1)*2);
                        // Si sposta all'inizio dell'headermeasure.
                        n += ((list[j]+1)*13);
                        // Legge dal file l'headermeasure del segnale
                        // selezionato.
                        read(handle,Headers+n,2*NWORD);
                       // Si sposta all'inizio dell'headervalue successivo.
                        n += 2*NWORD;
                          }
     #endif
}

void reorder(unsigned char *string)
{
        int n=Strlen(string)+1;
        unsigned char *tmp=new unsigned char [n];
        int i=0;
       int j=0;

        memcpy(tmp,string,n);
        do {
                switch(tmp[i]) {
                                        case IDML_TIME:
                                               i+=7;
                                               break;
                                        case IDML_VALUE:
                                               i+=3;
                                               break;
                                        case IDML_SUB:
                                        case IDML_COST:
                                            i += 2;
                                               break;
                                        case IDML_RANGE:
                                               memcpy(string+j,tmp+i,4);
                                               j+=4;
                                            i+=3;
                                               break;
                                        case IDML_NODIGIT:
                                               i++;
                                               break;
                                        case IDML_SIZE:
                                               i++;
                                            break;
                                        }
                    i++;
           } while(tmp[i] != IDML_END);

       i=0;
        do {
                switch(tmp[i]) {
                                        case IDML_TIME:
                                               memcpy(string+j,tmp+i,8);
```

```
                                        j += 8;
                                    i += 7;
                                        break;
                                case IDML_VALUE:
                                        memcpy(string+j,tmp+i,4);
                                        j += 4;
                                    i += 3;
                                        break;
                                case IDML_SUB:
                                case IDML_COST:
                                        memcpy(string+j,tmp+i,3);
                                        j += 3;
                                    i += 2;
                                        break;
                                case IDML_RANGE:
                                        i += 3;
                                        break;
                                case IDML_NODIGIT:
                                case IDML_SIZE:
                                        memcpy(string+j,tmp+i,2);
                                        j += 2;
                                    i += 1;
                                        break;
                                }
                        i++;
                    } while(tmp[i] != IDML_END);

                string[j]=IDML_END;
                delete tmp;
}

////////////////////////////////////////////////////////////////////
//                                                                //
//  La funzione Getbytemisura estrae la lunghezza della misura    //
//  dalla stringa di definizione.                                 //
//  La funzione ritorna la lunghezza della stringa di misura      //
//                                                                //
////////////////////////////////////////////////////////////////////
short int Getbytemisura(unsigned char *string)
{
                int i=0;

                do {
                        switch(string[i]) {
                                case IDML_TIME:
                                        i++;
                                        YEAR_OFFSET=(unsigned short) string[i];
                                        i++;
                                        YEAR_LENGTH=(unsigned short) string[i];
                                        i++;
                                        MONTH_OFFSET=(unsigned short) string[i];
                                        i++;
                                        DAY_OFFSET=(unsigned short) string[i];
                                        i++;
                                        HOUR_OFFSET=(unsigned short) string[i];
                                        i++;
                                        MIN_OFFSET=(unsigned short) string[i];
                                        i++;
                                        SEC_OFFSET=(unsigned short) string[i];
                                        break;
                                case IDML_VALUE:
                                        i+=3;
                                        break;
                                case IDML_SUB:
                                case IDML_COST:
                                    i += 2;
                                        break;
                                case IDML_RANGE:
                                        i += 3;
                                        break;
                                case IDML_NODIGIT:
                                        i++;
                                        break;
                                case IDML_SIZE:
                                        return(string[i+1]);
```

```
                                        }
                        i++;
                    } while(string[i] != IDML_END);
                return(0);
}

void Addbytemisura(unsigned char *string,int idml,int v1,int v)
{
        int i=0;

        do {
                switch(string[i]) {
                                        case IDML_TIME:
                                          i +=7 ;
                                          break;
                                        case IDML_VALUE:
                                          i+=3;
                                          break;
                                        case IDML_SUB:
                                        case IDML_COST:
                                          if (idml == IDML_COST && string[i+2] ==
v1) string[i+1] += v;
                                          i += 2;
                                          break;
                                        case IDML_RANGE:
                                          i += 3;
                                          break;
                                        case IDML_NODIGIT:
                                          i++;
                                          break;
                                        case IDML_SIZE:
                                          if (idml == IDML_SIZE) string[i+1] += v;
                                          i++;
                                          break;
                                        }

                        i++;
                    } while(string[i] != IDML_END);
                return;
}

void GetTime(unsigned char *misura,struct date &d,struct time &t)
{
        d.da_mon=GETMONTH(misura);
        d.da_day=GETDAY(misura);
        d.da_year=GETYEAR(misura);

        t.ti_hour=GETHOUR(misura);
        t.ti_min=GETMIN(misura);
        t.ti_sec=GETSEC(misura);
}

BOOL Aceas::OpenTest(char *sourcefile)
{
        n_test=0;
        q_test=0;
        end_file_test=FALSE;

        #ifdef _ACE_EVOLUTION
    memset(evolution_buffer,0,sizeof(evolution_buffer));
    memset(nevolution_buffer,0,sizeof(nevolution_buffer));
    memset(header_value_test,0,sizeof(header_value_test));
    memset(header_time_test,0,sizeof(header_time_test));
    memset(header_measures,0,sizeof(header_measures));
    #endif

        // Inizializza le variabili usate dalla funzione GET
        GR=0;
        BIT_R=0;

        // Azzera il checksum ed il numero di misure estratte
        header_aceas.crc=0;
        header_aceas.nmisure=0;
        // Apre il file di ingresso.
        source=open(sourcefile,O_BINARY | O_RDONLY);
```

```
        if (source == -1) return FALSE;

        #ifdef ENABLE_ACEAS_DEBUG
        char debugfile[128];

sprintf(debugfile,"%s%c%s%cace_test.txt",PROGRAMMDIRECTORY,DIR_CHAR,TMP_DIRECTORY,DI
R_CHAR);
        aceas_debug_handle=open((char*)debugfile, O_CREAT|O_WRONLY|O_TRUNC
                        #ifdef _LINUX
                        ,S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
                        #endif
                        );
        #endif // ENABLE_ACEAS_DEBUG

        // Legge l'intestazione del file aperto.
        q_test=read(source,(unsigned char *) &header_test,sizeof(header_test));
        if (q_test < (int)sizeof(header_test)) {
                                        close(source);
                                         source=0;
                                         return FALSE;
                                        }
        // Confronta la segnatura per verificare che si tratti
        // di un file compresso con la release 4 del algoritmo
        // ACE4.
        if (strcmp(header_test.signature,header_aceas.signature) != 0) {

//String[0]=IDML_END;
                                                                close(source);
                                                                source=0;
                                                                return(FALSE);
                                                                }

        // Legge il numero di misure dal file.
        if (!header_test.nmisure) {
                                close(source);
                                source=0;
                                return(FALSE);
                                   }
        read(source,String,header_test.leng_ml);
        #ifdef _ACE_EVOLUTION
                                     for          (i_test=0;i_test<nlist+1;i_test++)
header_value_test[i_test]=&evolution_buffer[i_test][0];
    #endif
        // Legge header_test e valori iniziali dal file compresso.
        LoadHeaders(source,header_time_test);

        // Acquisisce i puntatori alla tabella temporale e
        // all'header_test temporale.
        #ifndef _ACE_EVOLUTION
        table_time_test=GETPOINTER(0,ID_TABELLA,Headers,list);
        header_time_test=GETPOINTER(0,ID_HEADERTIME,Headers,list);
        // Compila la tabella temporale se ci sono più di una misura.
        if                  (header_test.nmisure                >                  1)
DECTABTIME(header_time_test,table_time_test,list[0]+1);
        #endif // _ACE_EVOLUTION

        // Memorizza il valore temporale iniziale per un utilizzo futuro.
    memcpy(binary1,header_time_test,HEADER_TIME_SIZE);

    #ifndef _ACE_EVOLUTION
    // Compila la tabella per ogni singolo segnale
        for (i_test=0;i_test<nlist;i_test++) {

table_value_test[i_test]=GETPOINTER(i_test,ID_TABELLA,Headers,list);
                header_value_test[i_test]=(unsigned                          short*)
GETPOINTER(i_test,ID_HEADERVALUE,Headers,list);

DECTAB(header_value_test[i_test],table_value_test[i_test],list[i_test]+1);

                // La lunghezza della prima tabella è incrementata
                // di 1 in quanto memorizza anche il segnale
                // temporale.
                if (!i_test) t_test=1; else t_test=0;
                // Calcola la lunghezza della codeword per ogni
```

```
                    // singolo segnale.
                    lbit[i_test]=LENG(table_value_test[i_test],list[i_test]+t_test+1);
    }
    #endif // _ACE_EVOLUTION

        // Estrae i valori iniziali del segnale temporale e di
        // ogni altro segnale dagli header_test.
        for (j_test=0;j_test<nlist;j_test++) {
                if  (!j_test)  INITFIRSTTEXTTIME((unsigned  char  *)  wr,(unsigned
short*) header_time_test);
                CONVERTNUMBERTOMEASURE((unsigned short*)
                        #ifndef _ACE_EVOLUTION
                        GETPOINTER(j_test,ID_MEASURE,Headers,list)
                        #else
                        &header_measures[j_test][0]
                        #endif
                        ,(unsigned char*) wr+offset[j_test],list[j_test]);
    }

        // Conserva la misura binaria per un utilizzo futuro.
        memcpy(previous,wr,bytemisura);

        WriteToBuffer(ACEAS_DEBUG_HANDLE,(unsigned char*) wr,bytemisura,FALSE);

        // Aggiorna il checksum locale.
        // Incrementa il numero di misure estratte.
        header_aceas.nmisure++;

        // Il valore di 'i' indica la quantità di buffer libero
        // in cui è possibile caricare informazione dal file.
        i_test=SWAPSIZE;
        // Se il numero di misure è minore di due la decompressione
        // si arresta.
        return TRUE;
}

BOOL Aceas::ScanTest(void)
{
        if (header_test.nmisure <= 1) {
            return TRUE;
        }
        // Il valore di n_test indica in questa posizione i_test numero di byte
        // presenti nel buffer di lettura.
        do {
        if (n_test < bytemisura && !end_file_test) {
                                        // Legge dal file tanti byte
                                        // quanti indicati dalla variabile
                                        // 'i_test'
                                        memcpy(bufferin,bufferin+q_test,n_test);
                                        q_test=0;

i_test=read(source,bufferin+n_test,i_test);
                                                        }
                                                        else
                                                        // indica che nessun byte
                                                        // può essere letto dal file
                                                        { i_test=-1; }
        // Da questa posizione i_test indica il numero di caratteri
        // letti dal file.
        // Se i_test == 0 la fine del file è stata raggiunta
        if (!i_test) end_file_test=TRUE;
        if (i_test == -1) {
                        // Nessun carattere è stato letto.
                        i_test = n_test;
                    }
                    else
                    {
                        // I caratteri letti vengono aggiunti a quelli
                        // presenti nel buffer.
                        i_test += n_test;
                    }

    #ifdef _ACE_EVOLUTION
    if (nevolution_buffer[nlist] > 1) {
            n_test=GET(header_value_test[nlist],delta_measure,list[nlist],(unsigned
```

```
short*) (bufferin+q_test),i_test);
            if (n_test > i_test) {
                        close(source);
                        source=0;
                        return(FALSE);
                       }
            q_test += i_test-n_test;
            if (n_test < bytemisura && !end_file_test) {
                memcpy(bufferin,bufferin+q_test,n_test);
                q_test=0;
                i_test=read(source,bufferin+n_test,SWAPSIZE-n_test);
                if (!i_test) end_file_test=TRUE;
                i_test += n_test;
                n_test = i_test;
                }
            i_test=n_test;
    } else memcpy(delta_measure,evolution_buffer[nlist][0].v,list[nlist]);
    GETTIME(binary1,(unsigned char*) wr,(unsigned short *) delta_measure);


    #endif
        // Il valore di 'i_test' indica da questo punto il numero di byte
        // presenti nel buffer di lettura.
        for (j_test=0;j_test<nlist;j_test++) {
                // Viene estratta la codeword dalla serie di bit
                // letta dal file. Ogni segnale possiede una codeword
                // differente.
                #ifdef _ACE_EVOLUTION
                if (nevolution_buffer[j_test] > 1) {
        n_test=GET(header_value_test[j_test],delta_measure,list[j_test]+1,(unsigned
short*) (bufferin+q_test),i_test);
        if (n_test > i_test) {
                                close(source);
                                source=0;
                                return FALSE;
                            }
                q_test += i_test-n_test;
                }                                                                  else
memcpy(delta_measure,evolution_buffer[j_test][0].v,list[j_test]+1);
                #else
                if (!lbit[j_test]) continue;
                n_test=GET((unsigned short*) (bufferin+q_test),(unsigned short*)
temp_buffer,lbit[j_test],i_test);
                // La funzione GET in caso di errore ritorna un valore
                // di byte presenti nel buffer superiore a quello
                // fornito.
                if (n_test > i_test) {
                                close(source);
                                source=0;
                                return FALSE;
                            }
                q_test += i_test-n_test;
                // Richiede il puntatore alla tabella del segnale corrente.
                // La tabella del primo segnale ingloba anche la tabella
                // temporale.
                if (!j_test) t_test=1; else t_test=0;
                // Trasforma la codeword in stringa binaria.
                DECOMPRESS(table_value_test[j_test],(unsigned       short       *)
temp_buffer,delta_measure,list[j_test]+t_test+1);
                #endif
                // Identifica il contenuto del buffer delta_measure.
                if (type[j_test] == IDML_DELAY) {
                                // Calcola la misura dalla derivata.

CALCMEASURE(previous+offset[j_test],delta_measure,(unsigned                 char*)
wr+offset[j_test],list[j_test],nodigit+n_nodigit-1,previous);
                                            }
                                            else
                                            {
                                // Copia la misura integralmente.

memcpy(wr+offset[j_test],delta_measure+1,list[j_test]);
                                            }
        #ifndef _ACE_EVOLUTION
                // Calcola il segnale temporale della misura.
```

```
                if        (!j_test)        GETTIME(binary1,(unsigned        char*)
wr,header_time_test,delta_measure[list[0]+1]);
                #endif
                // Legge altri byte dal file in ingresso per rifornire
                // il buffer di elaborazione. Il numero di byte letti
                // è puramente arbitrario.
                if (n_test < bytemisura && !end_file_test) {
                                        memcpy(bufferin,bufferin+q_test,n_test);
                                        q_test=0;

i_test=read(source,bufferin+n_test,SWAPSIZE-n_test);
                                        if (!i_test) end_file_test=TRUE;
                                        i_test += n_test;
                                        n_test = i_test;
                                }
                i_test=n_test;
                                }
        // Calcola il numero di byte necessari per completare il buffer
        // di ingresso.
        i_test=SWAPSIZE-n_test;
        // Memorizza la misura corrente per essere usata in coppia con la
        // successiva. L'inizializzazione del buffer avviene all'inizio
        // della funzione.
        memcpy(previous,wr,bytemisura);

        WriteToBuffer(ACEAS_DEBUG_HANDLE,(unsigned char *) wr,bytemisura,FALSE);

        // Incrementa il numero delle misure estratte.
        header_aceas.nmisure++;
        // Rivela la fine della decompressione.
        if (i_test == SWAPSIZE && end_file_test == TRUE && header_aceas.nmisure ==
header_test.nmisure) {
            return TRUE;
        }
        } while(n_test > bytemisura);
        return CONTINUE;
}

BOOL Aceas::CloseTest(void)
{
        // Chiude i file di ingresso e di uscita.
        close(source);
        // Libera la memoria allocata.
        source=0;
        #ifdef ENABLE_ACEAS_DEBUG
        close(aceas_debug_handle);
        #endif
        WriteToBuffer(ACEAS_DEBUG_HANDLE,(unsigned char*) wr,0,FALSE,TRUE);
        // Ritorna un errore se il checksum locale è differente da quello
        // memorizzato nel file compresso.
        if (header_aceas.crc != header_test.crc) return(FALSE);

        return(TRUE);
}

unsigned char *parser(unsigned char *string,char *measure)
{
        unsigned char constants[128];
        unsigned char times[8]={IDML_TIME,0,0,0,0,0,0,0};
        unsigned char fields[128];
        int h=0,k=0,n,m;
        int i=0,j=0;
        int itoken=0;
        char type=0;

        do {

                switch(measure[j]) {
                                case '.':
                                        constants[h++]=IDML_COST;
                                        constants[h++]=j;
                                        constants[h++]=measure[j];
                                        constants[h++]=IDML_NODIGIT;
                                        constants[h++]=j;
                                break;
```

```
                                        case DATE_SEPARE:
                                                if (!itoken || type == 0x20) { // Month
times[3]=itoken+(type?1:0);
                                                }
                                                else
                                                { // Day
                                                    times[4]=itoken+1;
                                                }
                                                itoken=j;
                                                type=measure[j];
                                                constants[h++]=IDML_COST;
                                                constants[h++]=j;
                                                constants[h++]=measure[j];
                                                break;
                                        case 0x20:
                                        case ']':
                                                switch(type){
                                                        case DATE_SEPARE: // Year
                                                            times[1]=itoken+1;
                                                            times[2]=j-itoken-1;
                                                            break;
                                                        case TIME_SEPARE: // Seconds
                                                            times[7]=itoken+1;
                                                            break;
                                                        case 0x20:
                                                        case 0x00:
                                                        case '[':
                                                            if (j-itoken-1) {
                                                            m=itoken+(type?1:0);
                                                            while((measure[m] >= '0'
&& measure[m] <= '9') || measure[m] == '.') m++;
                                                            if (m == j) {
                                                            fields[k++]=IDML_VALUE;
                                                            fields[k++]=IDML_DELAY;

fields[k++]=itoken+(type?1:0);

                                                            fields[k++]=j-itoken-
1+(type?0:1);
                                                                }
                                                                else
                                                                    if
(measure[itoken+1] >= 'a' && measure[itoken+1] <= 'z' && j-itoken-1 == 1) {

fields[k++]=IDML_VALUE;

fields[k++]=IDML_DELAY;

fields[k++]=itoken+1;

fields[k++]=j-itoken-1;

fields[k++]=IDML_RANGE;   // Nella versione fast IDML_RANGE deve stare

fields[k++]=itoken+1;     // prima di SUB

fields[k++]='d';

fields[k++]='a';

fields[k++]=IDML_SUB;

fields[k++]=itoken+1;

fields[k++]='a';
                                        }
                                                                                }
                                                            break;
                                                    }
                                                itoken=j;
                                                type=measure[j];
                                                constants[h++]=IDML_COST;
                                                constants[h++]=j;
                                                constants[h++]=measure[j];
                                                break;
```

```
                                        case TIME_SEPARE:
                                                switch(type){
                                                        case '+':
                                                                break;
                                                        case 0x20:          // Hour
                                                                times[5]=itoken+1;
                                                                break;
                                                        case TIME_SEPARE:  // Min
                                                                times[6]=itoken+1;
                                                                break;
                                                    }
                                                itoken=j;
                                                type=measure[j];
                                                constants[h++]=IDML_COST;
                                                constants[h++]=j;
                                                constants[h++]=measure[j];
                                                break;
                                        case 0x0A:
                                                constants[h++]=IDML_COST;
                                                constants[h++]=j;
                                                constants[h++]=measure[j];
                                                break;
                                        case 0x0D:
                                                if       (type       ==       TIME_SEPARE)
times[7]=itoken+1;
                                                                                    else
                                                                                {
                                                if (j-itoken-1) {
                                                        m=itoken+1;
                                                        while((measure[m]   >=   '0'   &&
measure[m] <= '9') || measure[m] == '.') m++;
                                                        if (m == j) {
                                                        fields[k++]=IDML_VALUE;
                                                        fields[k++]=IDML_DELAY;
                                                        fields[k++]=itoken+1;
                                                        fields[k++]=j-itoken-1;
                                                                    }
                                                                   else
                                                                  if (measure[itoken+1]
>= 'a' && measure[itoken+1] <= 'z' && j-itoken-1 == 1) {

fields[k++]=IDML_VALUE;

fields[k++]=IDML_DELAY;

fields[k++]=itoken+1;
                                                                                        fields[k++]=j-
itoken-1;

fields[k++]=IDML_RANGE;          // Nella versione fast IDML_RANGE deve stare

fields[k++]=itoken+1;            // prima di SUB

fields[k++]='d';

fields[k++]='a';

fields[k++]=IDML_SUB;

fields[k++]=itoken+1;

fields[k++]='a';

                                }
                                                                        }
                                                                    }
                                                constants[h++]=IDML_COST;
                                                constants[h++]=j;
                                                constants[h++]=measure[j];
                                                break;
                                        case 'p':
                                        case 'T':
                                        case '+':
                                        case '[':
                                                itoken=j;
```

```
                                        type=measure[j];
                                        constants[h++]=IDML_COST;
                                        constants[h++]=j;
                                        constants[h++]=measure[j];
                                        break;
                                default:
                                        if ((measure[j] < '0' || measure[j] > '9')
                                            &&
                                            (measure[j] < 'a' || measure[j] > 'z')
                                            &&
                                            (measure[j] < 'A' || measure[j] > 'Z')
                                            ) return NULL;
                                }
        } while(measure[j++] != 0x0A);

        n=j;

        for (j=0;j<8;j++) string[i++] = times[j];
        for (j=0;j<k;j++) string[i++] = fields[j];
        for (j=0;j<h;j++) string[i++] = constants[j];

        string[i++]=IDML_SIZE;
        string[i++]=n;
        string[i]=IDML_END;
        return string;
}

void print(FILE *handle,unsigned char *string)
{
        int i=0;

        if (!handle) return;

        do {
                switch(string[i]) {
                                case IDML_TIME:
                                        i++;
                                        fprintf(handle,"YEAR                OFFSET:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"YEAR                LENGTH:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"MONTH               OFFSET:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"DAY                 OFFSET:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"HOUR                OFFSET:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"MINUTE              OFFSET:
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"SECONDS             OFFSET:
%d\n",string[i]);
                                        break;
                                case IDML_VALUE:
                                        i++;
                                        switch(string[i]) {
                                                case IDML_DELAY:
                                                        fprintf(handle,"VALUE
DELAY\n");
                                                        break;
                                                case IDML_TOTAL:
                                                        fprintf(handle,"VALUE
TOTAL\n");
                                                        break;
                                                default:
                                                        fprintf(handle,"VALUE
NAN\n");
                                                        break;
                                                         }
                                        i++;
```

```
                                        fprintf(handle,"VALUE                OFFSET
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"VALUE                LENGTH
%d\n",string[i]);
                                        break;
                                    case IDML_SUB:
                                        i++;
                                        fprintf(handle,"SUB                  OFFSET
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"SUB                      %d
'%c'\n",string[i],string[i]);
                                        break;
                                    case IDML_COST:
                                        i++;
                                        fprintf(handle,"COST                 OFFSET
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"COST                     %d
'%c'\n",string[i],string[i]);
                                        break;
                                    case IDML_RANGE:            // Viene saltata.
                                        i++;
                                        fprintf(handle,"RANGE                OFFSET
%d\n",string[i]);
                                        i++;
                                        fprintf(handle,"RANGE        sup         %d
'%c'\n",string[i],string[i]);
                                        i++;
                                        fprintf(handle,"RANGE        inf         %d
'%c'\n",string[i],string[i]);
                                        break;
                                    case IDML_SIZE:                 // Acquisisce la
lunghezza
                                        i++;                      // della stringa di
misura
                                        fprintf(handle,"LENGTH %d\n",string[i]);
                                        break;
                                    case IDML_NODIGIT:
                                        i++;
                                        fprintf(handle,"NODIGIT %d\n",string[i]);
                                        break;
                                    }
                    i++;
                } while(string[i] != IDML_END);
 }

void  digitalizer(char  *string,char  *measure,unsigned  char  *list,unsigned  char
*offset,unsigned short nlist,unsigned short bytemisura,unsigned long *data)
{
        char *buffer=new char[bytemisura];
    unsigned short num[NWORD];
    unsigned long *p=(unsigned long*) num;

    memcpy(buffer,measure,bytemisura);
        TESTMISURA((unsigned     char     *)     buffer,(unsigned     char     *)
string,bytemisura,bytemisura);

        for (int j=0;j<nlist;j++) {
                            CONVERTMEASURETONUMBER((unsigned                    char*)
buffer+offset[j],num,list[j]);
                            data[j]=p[0];
                                }

        delete[] buffer;
}

BOOL Aceas::Init(unsigned char* filename)
{
    // Alloca la memoria per i buffer
    // di ingresso uscita.
    //header_aceas.nfile=nfile;
    //header_aceas.nmisfile=nmisfile;
    bufferin=new unsigned char[SWAPSIZE];
```

```
    if (bufferin == 0) return(FALSE);
    bufferout=new unsigned char[SWAPSIZE];
    if (bufferout == 0) {
                delete[](bufferin);
                return(FALSE);
                }

    if (filename) {
        char model[256];

        // Apre il file di ingresso
        int handle=open((char*) filename,O_BINARY | O_RDONLY);
            // Ritorna FALSE se il file non è stato trovato
        if (handle == -1) return(FALSE);

        read(handle,model,sizeof(model));
        parser(String,model);
        bytemisura=Getbytemisura(String);
        lseek(handle,0,SEEK_SET);
        close(handle);
        // Alloca la memoria per l'elaborazione del segnale.
        if (AllocHeaders(String) == FALSE) { close(handle); return FALSE; }
    }
    return(TRUE);
}

Aceas::Aceas(void)
{
    // Inizializza le variabili
    temp_buffer=0;
    initscan=FALSE;
    initcomp=FALSE;
    pwrite=0;
    strcpy(header_aceas.signature,SIGNATURE);
    drain=0;
    tmp=0;
    source=0;
}
```

APPENDICE A5

Appendice A5 – Esempio di codice applicativo

```
/*
  Sistema di compressione ACEAS studiato per la versione network
  del sistema MagNet
  Antonino Sicali (c) 1998-2019
  Istituto Nazionale di Geofisica e Vulcanologia
  Catania
*/

#include "../config.h"
#include <stdio.h>
#ifdef _ACE_EVOLUTION
#include "aceas.evolution.h"
#include "ace.evolution.h"
#else
#include "aceas.h"
#include "ace.h"
#endif
#ifdef _LINUX
#include <unistd.h>
#endif

//#define _DISABLE_COMPRESSION

#define SENSORS_DATA  3
#define SENSORS_DIGIT 4
#define SENSORS_SPACE 1
#define SENSORS_MINIMO_MISURE 5

#define NOINIT   "\nErrore durante l'inizializzazione della classe"
#define ERRCOMP  "\nErrore durante la compressione"
#define ERRDEC   "\nErrore durante la decompressione"
#define NOSIGNAL "\nFile privo di misure"
#define INFILE   "1day.txt"
#define OUTFILE  "1day.dat"
#define TEMPFILE "1day.ace"

char PROGRAMMDIRECTORY[1024];

#ifdef __WINDOWS__
void setProgrammDirectory(void)
{
    int i=GetModuleFileName(NULL,PROGRAMMDIRECTORY,sizeof(PROGRAMMDIRECTORY)-1);
        i--;
        while(i >= 0 && PROGRAMMDIRECTORY[i] != '\\') i--;
        PROGRAMMDIRECTORY[i]=0;
}
#elif defined(_LINUX)
bool setProgrammDirectory(void)
{

   int i = readlink("/proc/self/exe", PROGRAMMDIRECTORY, sizeof(PROGRAMMDIRECTORY));

    if (i < 0) return false;

    while(i >= 0 && PROGRAMMDIRECTORY[i] != '/') i--; //system dir
    i--;
    while(i >= 0 && PROGRAMMDIRECTORY[i] != '/') i--;
    PROGRAMMDIRECTORY[i]=0;
    return true;
}
#endif

unsigned char DefMeasure[128];

Aceas ace4;               // dichiarazione della classe
                             // per la compressione
                             // decompressione

#define DATE_SEPARE '-'
#define TIME_SEPARE ':'

// I parametri formali argc e argv non vengono usati dalla funzione e
// sono commentati per ridurre a zero i warning
int main(int /*argc*/,char */*argv*/[])
```

```
{
        int r;
        char infile[128];
        char tempfile[128];
        char outfile[128];

        #ifdef _LINUX
       setvbuf(stdout, NULL, _IONBF, 0);
      #endif

        printf("\nstart\n");

    #ifdef _LINUX
        if (!
    #endif
     setProgrammDirectory()
    #ifdef _LINUX
     )
     return -1;
    #else
    ;
    #endif // _LINUX

    sprintf(infile,"%s%c%s",PROGRAMMDIRECTORY,DIR_CHAR,INFILE);
    sprintf(outfile,"%s%c%s",PROGRAMMDIRECTORY,DIR_CHAR,OUTFILE);
    sprintf(tempfile,"%s%c%s",PROGRAMMDIRECTORY,DIR_CHAR,TEMPFILE);

        // inizializza la classe
        // Non cancellare, necessario per la decompressione
        if (ace4.Init() == FALSE) { printf(NOINIT); return -1; }

        // funzione di compressione
        #ifndef _DISABLE_COMPRESSION
        r=ace4.Compress(infile,tempfile,0);
        char tmp[128];

sprintf(tmp,"%s%c%s%cstring.txt",PROGRAMMDIRECTORY,DIR_CHAR,TMP_DIRECTORY,DIR_CHAR);
        FILE *handle=fopen(tmp,"wt+");
        print(handle,ace4.String);
        fclose(handle);
        if (!r) { printf("Error on compress\n"); return -1; }
        // in caso di errore sul file ritorna errore
        if (r == FALSE) { printf(ERRCOMP); return; }
        #endif


        // funzione di decompressione
        r=ace4.Decompress(tempfile,outfile);
        switch(r) {
                        case FALSE:
                              // presenza di errori nel file
                              printf(ERRDEC);
                              break;
                        case NOMEASURE:
                              // il file è vuoto
                              printf(NOSIGNAL);
                              break;
                  }
        printf("Error on station:%ld\n",ace4.GetError());
        if (ace4.AllocHeaders(ace4.String) == FALSE) { printf("NO MEMORY FREE\n");
return -1; }
        if (ace4.Init() == FALSE) { printf("NO MEMORY FREE\n"); return -1; }
        if (!ace4.OpenTest(tempfile)) { printf("[err1]\n"); return -1; }
        while (1) {
                r=ace4.ScanTest();
                if (r == TRUE) break;
                if (r == FALSE) { printf("[err2]\n"); return -1; }
                  }
        if (!ace4.CloseTest()) { printf("[err3]\n"); return -1; }
        printf("[no error]\n");
        return 0;
}
```

QUADERNI di GEOFISICA

ISSN 1590-2595 http://istituto.ingv.it/it/le-collane-editoriali-ingv/quaderni-di-geofisica.html/

I QUADERNI DI GEOFISICA (QUAD. GEOFIS.) accolgono lavori, sia in italiano che in inglese, che diano particolare risalto alla pubblicazione di dati, misure, osservazioni e loro elaborazioni anche preliminari che necessitano di rapida diffusione nella comunità scientifica nazionale ed internazionale. Per questo scopo la pubblicazione on-line è particolarmente utile e fornisce accesso immediato a tutti i possibili utenti. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi. I QUADERNI DI GEOFISICA sono presenti in "Emerging Sources Citation Index" di Clarivate Analytics, e in "Open Access Journals" di Scopus.

QUADERNI DI GEOFISICA (QUAD. GEOFIS.) welcome contributions, in Italian and/or in English, with special emphasis on preliminary elaborations of data, measures, and observations that need rapid and widespread diffusion in the scientific community. The on-line publication is particularly useful for this purpose, and a multidisciplinary Editorial Board with an accurate peer-review process provides the quality standard for the publication of the manuscripts. QUADERNI DI GEOFISICA are present in "Emerging Sources Citation Index" of Clarivate Analytics, and in "Open Access Journals" of Scopus.

RAPPORTI TECNICI INGV

ISSN 2039-7941 http://istituto.ingv.it/it/le-collane-editoriali-ingv/rapporti-tecnici-ingv.html/

I RAPPORTI TECNICI INGV (RAPP. TEC. INGV) pubblicano contributi, sia in italiano che in inglese, di tipo tecnologico come manuali, software, applicazioni ed innovazioni di strumentazioni, tecniche di raccolta dati di rilevante interesse tecnico-scientifico. I RAPPORTI TECNICI INGV sono pubblicati esclusivamente on-line per garantire agli autori rapidità di diffusione e agli utenti accesso immediato ai dati pubblicati. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi.

RAPPORTI TECNICI INGV (RAPP. TEC. INGV) publish technological contributions (in Italian and/or in English) such as manuals, software, applications and implementations of instruments, and techniques of data collection. RAPPORTI TECNICI INGV are published online to guarantee celerity of diffusion and a prompt access to published data. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

MISCELLANEA INGV

ISSN 2039-6651 http://istituto.ingv.it/it/le-collane-editoriali-ingv/miscellanea-ingv.html

MISCELLANEA INGV (MISC. INGV) favorisce la pubblicazione di contributi scientifici riguardanti le attività svolte dall'INGV. In particolare, MISCELLANEA INGV raccoglie reports di progetti scientifici, proceedings di convegni, manuali, monografie di rilevante interesse, raccolte di articoli, ecc. La pubblicazione è esclusivamente on-line, completamente gratuita e garantisce tempi rapidi e grande diffusione sul web. L'Editorial Board INGV, grazie al suo carattere multidisciplinare, assicura i requisiti di qualità per la pubblicazione dei contributi sottomessi.

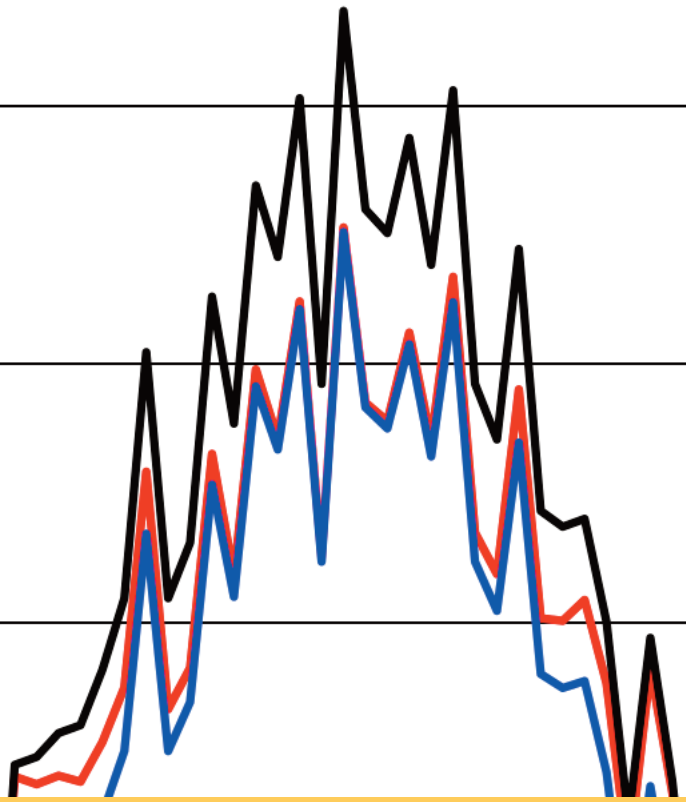
MISCELLANEA INGV (MISC. INGV) favours the publication of scientific contributions regarding the main activities carried out at INGV. In particular, MISCELLANEA INGV gathers reports of scientific projects, proceedings of meetings, manuals, relevant monographs, collections of articles etc. The journal is published online to guarantee celerity of diffusion on the internet. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

Coordinamento editoriale e impaginazione
Francesca DI STEFANO, Rossella CELI
Istituto Nazionale di Geofisica e Vulcanologia

Progetto grafico e impaginazione
Barbara ANGIONI
Istituto Nazionale di Geofisica e Vulcanologia

Istituto Nazionale di Geofisica e Vulcanologia
Via di Vigna Murata, 605
00143 Roma
tel. +39 06518601

www.ingv.it



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA